SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni diplomski studij

REALIZACIJA GRAFIČKOG KORISNIČKOG SUČELJA ZA TV APLIKACIJU ELEKTRONSKI PROGRAMSKI VODIČ

Diplomski rad

Mijo Vračević

Osijek, 2016.

SADRŽAJ

1. UVOD

Brzi razvoj digitalne tehnologije također je zahvatio i televizijsku industriju. Prelazak s analognih na digitalne TV sustave posljednjih godina prisutan je u cijelom svijetu i donosi sa sobom brojne prednosti i nove mogućnosti. Razloga za prelazak na digitalne sustave ima napretek; kvalitetniji prijenos slike i zvuka, bolje iskorištenje radio-frekvencijskog spektra, otpornost na smetnje, veća mogućnog obrade signala, interaktivnost, itd. Ponuda postaje sve raznovrsnija, a korisnici sve zahtjevniji. Uz veći broj kanala i sadržaja, korisnici očekuju nove digitalne TV usluge, te integraciju multimedijskih i Internet tehnologija.

U sklopu ovog diplomskog rada izrađena je jedna TV aplikacija koju nudi digitalna televizija – Elektronski programski vodič (EPG, engl. *Electronic program guide*). EPG praktički zamjenjuje teletekst i pruža ljepši prikaz informacija o trenutnim i nadolazećim događajima. Aplikacija omogućava tablični prikaz EPG događaja za dostupne kanale i dane, navigaciju kroz EPG korištenjem tipkovnice, filtriranje događaja po žanrovima, prikaz dodatnih informacija za odabrani događaj te dinamičko osvježavanje prikazanih EPG događaja. Ciljana platforma je uređaj s Linux operacijskim sustavom.

U drugom poglavlju rada izložene su osnovne informacije o digitalnoj televiziji i standardima digitalne televizije koji se koriste u svijetu, te način slanja informacija u digitalnoj televiziji. U trećem poglavlju objašnjen je Qt radni okvir u kojem je izrađena aplikacija, dok je u četvrtom poglavlju opisan sam način realizacije i funkcionalnost aplikacije. Zadnje, peto poglavlje, donosi zaključke.

2. DIGITALNA TELEVIZIJA

Analogna tehnologija odašiljanja televizijskih kanala nije mogla pratiti potražnju korisnika i veliki rast broja televizijskih kanala te je tako došlo do zagušenja radiofrekvencijskog spektra namijenjenog radiodifuziji. Digitalni sustav odašiljanja i komprimiranja slike omogućio je odašiljanje više televizijskih programa u jednom radiofrekvencijskom kanalu i niz drugih mogućnosti. Razni postupci kodiranja i kompresije omogućavaju da se u jednaku širinu kanala kao u analognoj tehnologiji smjesti veći broj televizijskih kanala. Uz veći broj kanala povećana je kvaliteta slike i zvuka s većom otpornošću na miješanje s ostalim signalima. Kod digitalne televizije slika i zvuk su jednaki na prijemu i izvoru emitiranja za razliku od analogne televizije gdje se šum značajno odražava na reprodukciju slike i zvuka. S pojavom digitalne televizije pojavile su se i raznovrsne usluge kao što su interaktivna televizija, digitalni teletekst, EPG. Neki od nedostataka digitalne televizije su zamjena postojeće analogne infrastrukture koja treba biti izmijenjena te se tako stvara trošak zbog postavljanja novih odašiljača i kupovine novih prijemnika. Također zbog uštede propusnog opsega prilikom kompresije događa se degradacija slike te je moguće da se dogodi i preljev boja, zamagljena slika, efekt stvaranja blokova, posterizacija, efekt digitalne litice itd.

U nekim zemljama se trenutno odvija tranzicija s DVB-T (engl. *Digital Video Broadcasting Terrestrial*) na DVB-T2, DVB standard druge generacije, a zemlje koje tek prelaze s analognih na digitalne sustave odmah primjenjuju DVB-T2. DVB-T2 standard koristi nove modulacije, bolje je iskorištenje spektra, povećana je robusnost te nudi nove programe i usluge korisnicima.

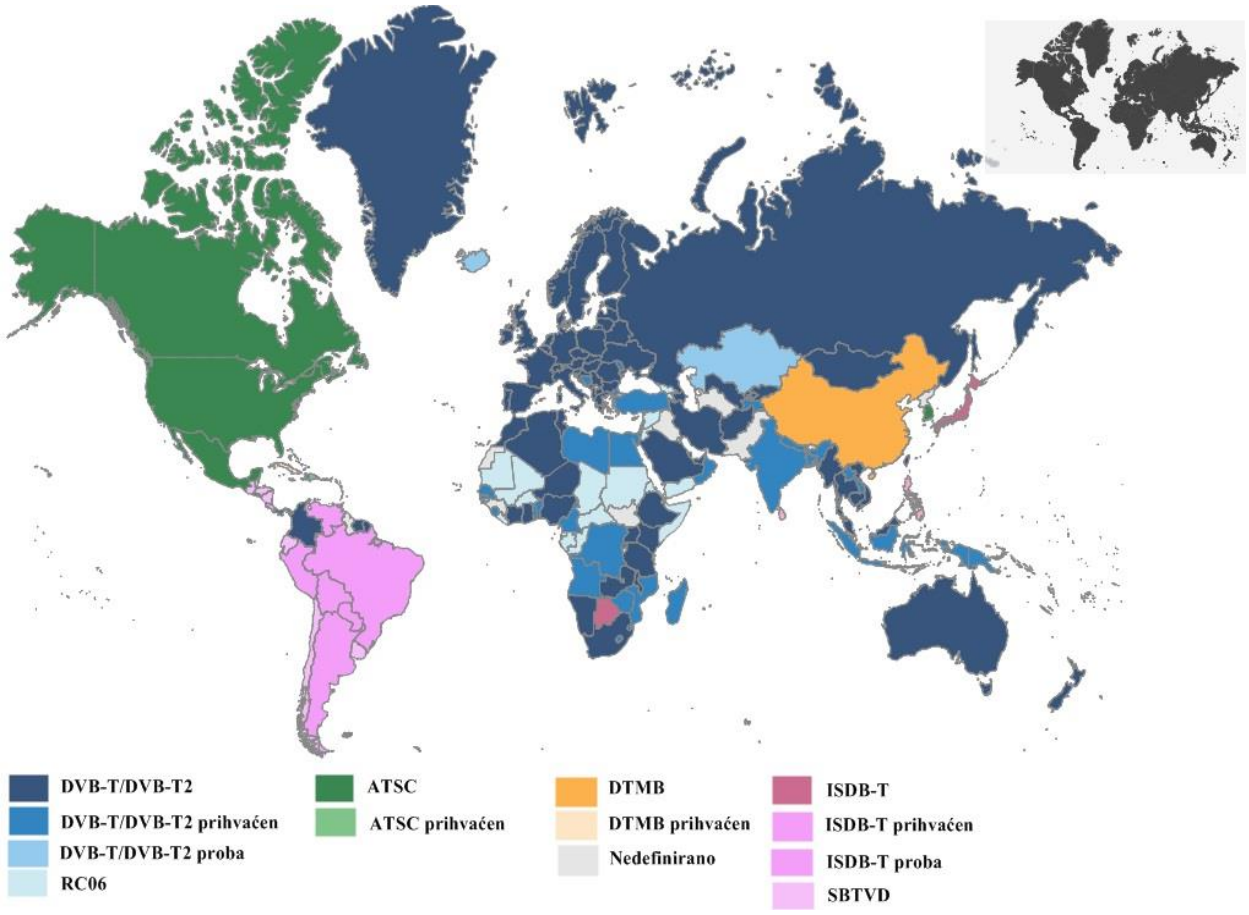
2.1. Standardi digitalne televizije

U svijetu se koristi nekoliko standarda digitalne televizije [2]:

- DVB - *Digital Video Broadcasting,*
- ATSC - *Advanced Television System Committee,*
- ISDB - *Integrated Services Digital Broadcasting,*
- DTMB - *Digital Terrestrial Multimedia Broadcasting,*
- SBTVD - *Brazilian Digital Television System,*

- ISDB - *Integrated Services Digital Broadcasting.*

Na slici 2.1. je prikazana karta svijeta gdje su zemlje označene bojom koja predstavlja pojedini standard.



Sl. 2.1. *Lokacije korištenja digitalnih televizijskih standarda u svijetu* [2]

2.1.1. DVB standard

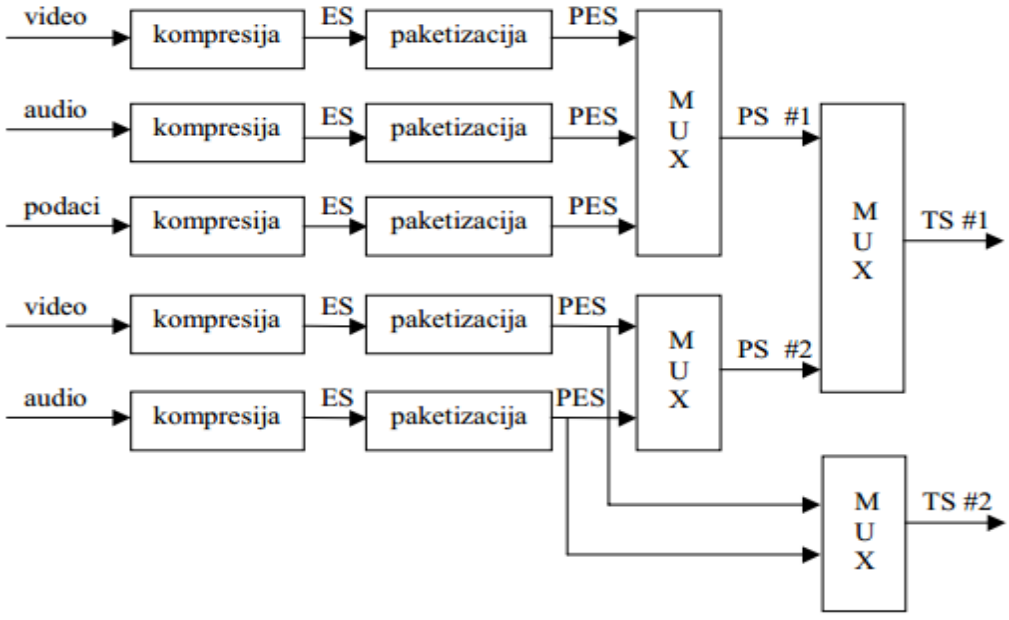
DVB standard za video kompresiju koristi MPEG-2 (engl. *Moving Picture Experts Group*) i MPEG-4, dok za audio kompresiju koristi MPEG-2, AC3 (engl. *Audio Compression*) i AAC (engl. *Advanced Audio Coding*). Jedan satelitski DVB transponder propusnog opsega 38 Mb/s pruža prijenos 4-8 standardnih kanala, 2 HDTV (engl. *High-definition television*) kanala, 150

radio kanala, 550 ISDN (engl. *Integrated Services Digital Network*) kanala. Najvažniji DVB standardi koji se koriste pri prijenosu signala digitalne televizije su [1]:

- DVB-S – DVB *Satellite* – definira satelitski prijenos DTV,
- DVB-C – DVB *Cable* – definira DTV prijenos putem digitalne kabelske mreže,
- DVB-T – DVB *Terrestrial* – definira zemaljski DTV prijenos putem UHF/VHF,
- DVB-H – DVB *Handheld* – DTV za prijenosne uređaje kao što su mobilni telefoni.

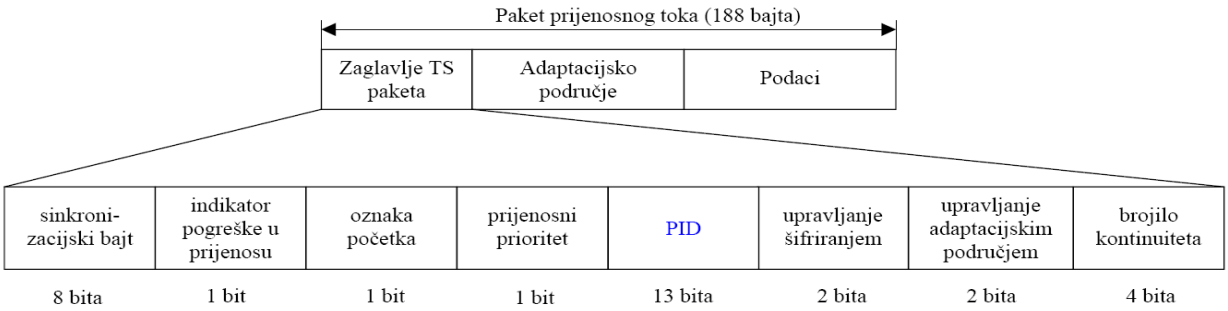
2.1.2. MPEG-2 norma

MPEG-2 norma koristi različite formate za multipleksiranje MPEG-2 elementarnih tokova u jedinstven informacijski tok podataka [1]. Multipleksiranje omogućava zajednički prijenos audio i video signala. Ako prijenosni medij nije podložan greškama u toku prijenosa, MPEG-2 elementarni tokovi se multipleksiraju u PS (engl. *Program Stream*). U suprotnom se elementarni tokovi multipleksiraju kao TS (engl. *Transport Stream*). TS je pogodan za kombiniranje više TV programa u jedan informacijski tok, a u toku postoji više elementarnih tokova – ES (engl. *Elementary Strem*), koji se kasnije paketiziraju u PES (engl. *Packetised Elementary Stream*). Paketi nisu jednoliko raspoređeni, a razlikuju se po svojim oznakama – PID (engl. *Packet Identifier*). Postupak multipleksiranja u jedinstveni informacijski tok prikazan je na slici 2.2.



Sl. 2.2. *Postupak multipleksiranja tokova kod MPEG-2* [1]

Na slici 2.3. prikazana je struktura MPEG-2 TS paketa.



Sl. 2.3. *Struktura MPEG-2 TS paketa* [1]

Prijemnik iz TS izdvaja podatke koji pripadaju određenom kanalu na temelju PID vrijednosti paketa, a povezivanje PID vrijednosti s odgovarajućim DTV kanalima omogućavaju kontrolni tokovi koji sadrže signalne tablice.

2.2. DVB signalne informacije

U toku se nalaze signalne tablice koje omogućuju povezivanje PID vrijednosti s određenim DTV kanalima sadržanim u prijenosnom toku. One se prenose kao zasebni tok multipleksiran unutar prijenosnog toka. U nastavku su nabrojane MPEG-2 tablice te SI DVB tablice.

MPEG-2 signalne tablice [3]:

- PAT (engl. *Program Association Table*)
  → Uvijek se šalje sa PID-om koji ima vrijednost 0x0000
  → Sadrži listu PID vrijednosti TS paketa s PMT tablicama
- PMT (engl. *Program Map Table*)
  → Lista PID vrijednosti TS paketa koji sadrže PES pakete pridružene određenom programu
  → Predefinirana PID vrijednost 0x0002
- NIT (engl. *Network Information table*)
  → Informacije o mreži koja emitira TS multipleks
  → Predefinirana PID vrijednost 0x0010
- CAT (engl. *Conditional Access Table*)
  → Definira tip kriptiranja i PID vrijednosti TS paketa koji sadrže informacije neophodne za dekriptiranje sadržaja
  → Predefinirana PID vrijednost 0x0001

- DCM-CC (engl. *Digital Storage Media Command and Control*)
  → kontrola prijema

DVB signalne tablice [3]:

- BAT (engl. *Bouquet Association Table*)
  → Kanali grupirani po cjelinama
  → Predefinirana PID vrijednost 0x0011
- SDT (engl. *Service Description Table*)
  → Naziv i detalji o kanalu
  → Predefinirana PID vrijednost 0x0011
- TDT (engl. *Time and Date Table*)
  → Informacija o vremenu emitiranja kanala
  → Predefinirana PID vrijednost 0x0014
- EIT (engl. *Event Description Table*)
  → Detalji o rasporedu emitiranja programa
  → Predefinirana PID vrijednost 0x0012

U ovom radu EIT tablica je od najvećeg značaja za aplikaciju budući da sadrži informacije o pojedinoj emisiji kao što su ime emisije, opis emisije, trajanje, početak i kraj emisije, itd. Postoje četiri vrste EIT tablica, koje se razlikuju na temelju *table_id* [3]:

1. EIT tablica koja nosi informacije o trenutnim/nadolazećim emisijama na trenutnom podatkovnom toku → *table_id* = "0x4E"
2. EIT tablica koja nosi informacije o trenutnim/nadolazećim emisijama na drugom podatkovnom toku → *table_id* = "0x4F"
3. EIT tablica koja nosi informacije o emisijama (po vremenu) na trenutnom podatkovnom toku → *table_id* = "0x50 - 0x5F"
4. EIT tablica koja nosi informacije o emisijama (po vremenu) na drugom podatkovnom toku → *table_id* = "0x60 - 0x6F"

Tablica 2.1. prikazuje format EIT tablice.

Tab. 2.1. *EIT tablica* [3]

Sintaksa	Broj bitova	Identifikator
event_information_section(){		
table_id	8	uimsbf
section_syntax_indicator	1	bslbf
reserved_future_use	1	bslbf
reserved	2	bslbf
section_length	12	uimsbf
service_id	16	uimsbf
reserved	2	bslbf
version_number	5	uimsbf
current_next_indicator	1	bslbf
section_number	8	uimsbf
last_section_number	8	uimsbf
transport_stream_id	16	uimsbf
original_network_id	16	uimsbf
segment_last_section_number	8	uimsbf
last_table_id	8	uimsbf
for(i=0;i<N;i++){		
event_id	16	uimsbf
start_time	40	bslbf
duration	24	uimsbf
running_status	3	uimsbf
free_CA_mode	1	bslbf
descriptors_loop_length	12	uimsbf
for(i=0;i<N;i++){		
descriptor()		
}		
}		
CRC_32	32	rpchof
}		

Slijedi opis polja iz tablice 2.1:

- *service_id* služi kao identifikator kanala, odnosno govori kojem kanalu pripada sekcija tablice,
- *section_length* govori od koliko se bajta sastoji sekcija EIT tablice gdje se nalazi,
- *version_number* predstavlja broj verzije podtablice,
- *current_next_indicator* govori da li je podtablica važeća ili ne - vrijednost 1 ili 0,
- *section_number* označava broj sekcije,
- *last_section_number* označava broj zadnje sekcije podtablice,
- *transport_stream_id* služi kao oznaka za identifikaciju TS-a,
- *original_network_id* oznaka za identifikaciju mreže koja vrši isporuku DTV sadržaja,

- *last_table_id* identificira zadnju korištenu vrijednost polja table_id,
- *event_id* je identifikator emisije,
- *start_time* je početno vrijeme emisije,
- *duration* sadrži trajanje emisije,
- *running_status* označava status emisije (npr. emisija je u toku),
- *free_CA_mode* je jednobitno polje koje označava da li su tokovi podataka zaključani ili ne – vrijednost 0 ili 1,
- *descriptors_loop_length* označava dužinu tablice s informacijama o emisiji,
- *CRC_32* označava polje za provjeru.

U *short event* deskriptoru se nalazi ime emisije te kratki opis emisije. Format deskriptora je prikazan u tablici 2.2.

Tab. 2.2. *Short event deskriptor* [3]

Sintaksa	Broj bitova	Identifikator
short_event_descriptor(){		
descriptor_tag	8	uimsbf
descriptor_length	8	uimsbf
ISO_639_language_code	24	bslbf
event_name_length	8	uimsbf
for (i=0;i<event_name_length;i++){		
event_name_char	8	uimsbf
}		
text_length	8	uimsbf
for (i=0;i<text_length;i++){		
text_char	8	uimsbf
}		
}		

2.3. Elektronski programski vodič

Elektronski programski vodič je aplikacija namijenjena za prikazivanje informacija o trenutnim i nadolazećim događajima na televizijskim kanalima. Izgleda kao izbornik (engl. *Menu-based system*) koji pokazuje liste događaja na dostupnim kanalima (televizijskim kanalima). Aplikacija je interaktivna i tako omogućuje korisnicima da koriste brojne funkcionalnosti. Korisnik može odabrati između dva izgleda prikazivanja događaja:

- izbornik koji prikazuje trenutni i nadolazeći događaj kanala,

- izbornik koji prikazuje događaje određenog vremenskog razdoblja (engl. *Timeline schedule*).

EPG aplikacija nudi korisniku filtriranje događaja po žanru i vremenu, navigaciju kroz događaje, označavanje istih te prikazivanje opisa događaja (informacije kao što su trajanje, žanr, razina roditeljske kontrole, opis komponenti događaja) ukoliko je opis dostupan.

3. KORIŠTENI PROGRAMSKI ALATI

3.1. Qt Framework

Qt je višeplatformski radni okvir (engl. *framework*) široko korišten za razvijanje softvera na desktop sustavima, ugrađenim sustavima te mobilnim sustavima. Razvoj Qt-a započeli su Haavard Nord i Eirik Chambe-Eng 1990. godine. 1994. godine osnovali su firmu po imenu Trolltech koja je pružala usluge prodavanja licenci i pružanja podrške. Firma Trolltech je tokom godina doživjela nekoliko akvizicija. Nakon prve akvizicije 2008. godine od strane Nokie, firma Trolltech je preimenovana u Qt Software, a zatim u Qt Development Frameworks. Danas se firma zove Qt Company i u vlasništvu je finske softverske firme Digia Plc, koja je 2012. preuzela Qt od Nokie. Qt 5 je posljednja realizirana verzija, a trenutno je aktivna Qt 5.7. Qt koriste brojne firme, a neke od njih su: LG, Panasonic, Sky, Thales, ABB, Rimac Automobili, Sennheiser, AMD, Navico, AutoIO, Harman, itd.

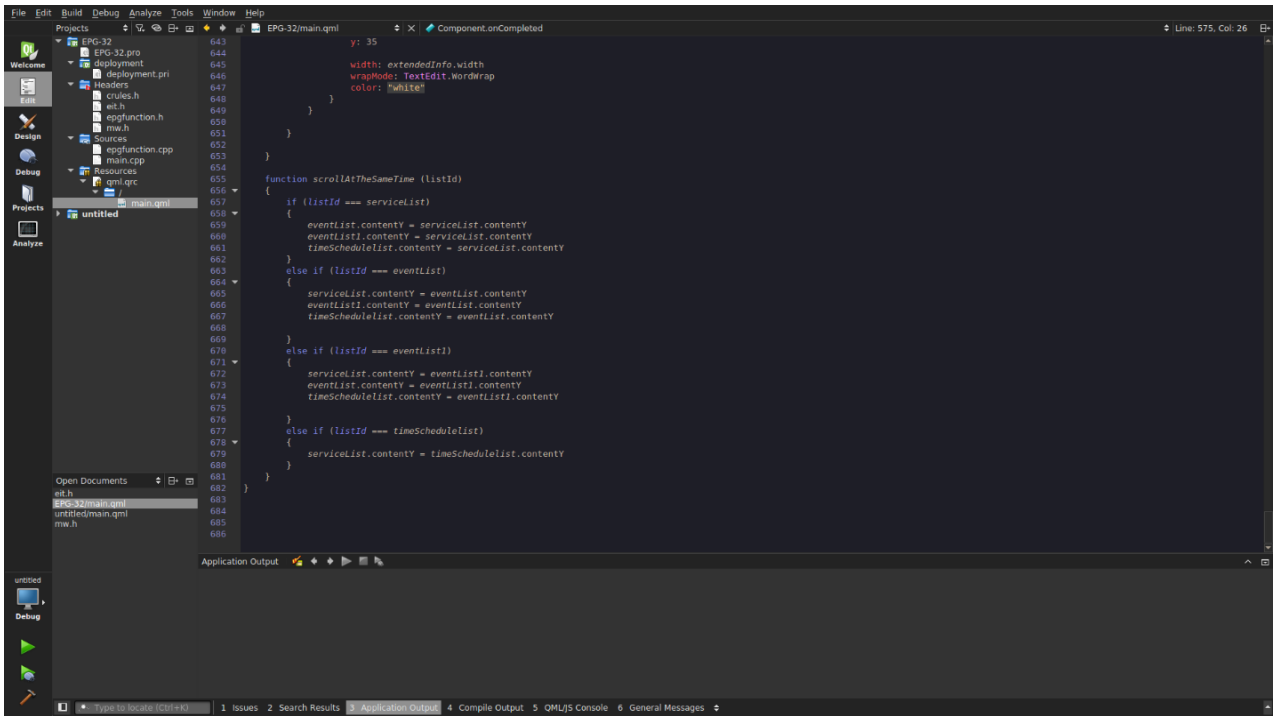
Može se koristiti za razvoj aplikacija na nekoliko ciljanih platformi kao što su:

- Desktop sustavi: Linux, OS X, Windows.
- Ugrađeni sustavi i stvarnovremenski sustavi: Linux, QNX, VxWorks, Windows.
- Mobilni sustavi: Android, iOS, Windows.

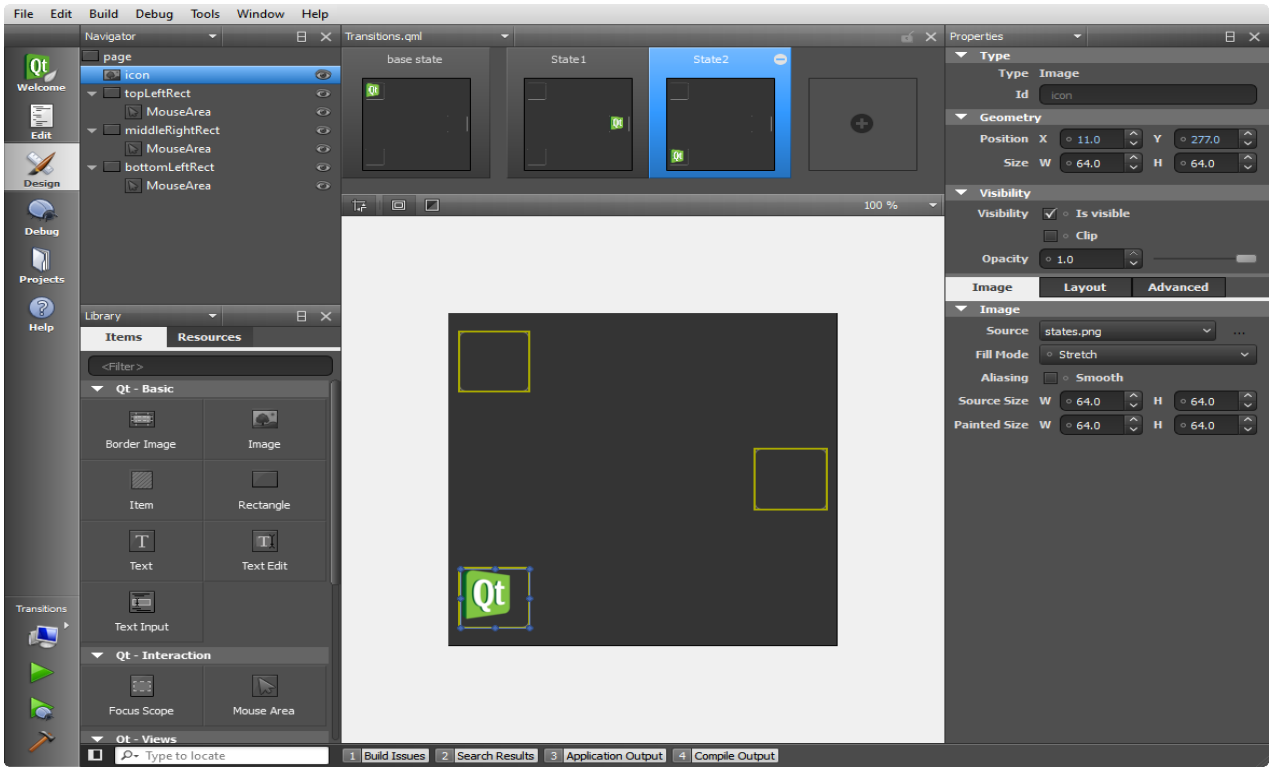
Qt se bazira na C++ programskom jeziku, a moguće ga je povezati i s raznim drugim programskim jezicima. U kombinaciji sa C++ programskim jezikom i Qt objektnim modelom omogućava kreiranje brzih i fleksibilnih grafičkih korisničkih sučelja (engl. *Graphical User Interface*, GUI). Mnoge značajke koje olakšavaju rad i nude brojne mogućnosti, od kojih će neke biti korištene i opisane u ovom radu, Qt provodi nasljeđivanjem *QObject* klase. *QObject* klasa je osnovna klasa svih Qt objekata i srce je Qt objektnog modela.

3.1.1. Qt alati

Qt *Creator* je IDE (engl. *Integrated development environment*) koji omogućava pisanje koda (Slika 3.1.), dok Qt *Designer* olakšava vizualizaciju gotovih ili napisanih elemenata(Slika.3.2.).



Sl.3.1. *Qt CreatorIDE*



Sl. 3.2. *Qt Designer*

3.1.2. QML

QML (engl. *Qt Meta Language*) je deklarativni jezik koji omogućuje opisivanje korisničkog sučelja u smislu vizualnih komponenti te integracije i odnosa između istih. To je

vrlo čitljiv jezik koji je osmišljen kako bi stvorene komponente bilo lako ponovno iskoristiti i prilagoditi. Koristeći *QtQuick* modul, dizajneri i programeri mogu lako kreirati korisničko sučelje u QML-u, i imaju mogućnost povezivanja korisničkog sučelja s programskim kodom napisanim u C++ programskom jeziku.

3.1.3. Qmake

Qmake je alat koji pojednostavljuje proces izrade projekta za različite platforme pomoću automatskog generiranja *makefile-a*. Generiranje se temelji na informacijama koje sadrži projektna datoteka sa ekstenzijom .pro. Modifikacija .pro datoteke potrebna je kako bi se spojili svi gradivni dijelovi aplikacije. U nastavku je prikazan primjer .pro datoteke:

```
// myproject.pro
SOURCES += main.cpp
```

Ako projekt koristi QML potrebno je uključiti *QtQuick* i *QtQml* modul:

```
QT += qml quick
```

Qmake koristi "=", "+=", "-=" za pridruživanje, dodavanje i uklanjanje elemenata. Sintaksa pokretanja qmake datoteke pomoću komandne naredbe je sljedeća:

```
qmake [mode] [options] files
```

Ponašanje projekta je moguće prilagođavati, a detaljnije podešavanje *Qmake* datoteke je dostupno u dokumentaciji [5].

3.1.4. Integracija: C++ i QML

QML jezik dizajniran je tako da se lako može proširiti sa C++ kodom. Klase u Qt QML modulu omogućuju QML objektima učitavanje i upravljanje pomoću C++ koda. Integracija s Qt meta-objekt sustavom omogućuje direktno pozivanje C++ funkcija iz QML-a. Na taj način omogućena je izrada interaktivnih aplikacija implementiranih s QML, JavaScript i C++ kodom. Integracija QML-a i C++ koda pruža niz mogućnosti [5]:

- odvajanje korisničkog sučelja od same logike aplikacije tako što je korisničko sučelje implementirano korištenjem QML i JavaScript koda, a logika pomoću C++ koda,
- pozivanje funkcionalnosti (funkcije implementirane u C++ modulu) iz QML-a (npr. Korištenje podatkovnog modela implementiranog u C++ kodu ili pozivanje funkcija u *third-party* C++ biblioteci),
- implementacija vlastitih QML objekt tipova u C++ dijelu.

Pružanje C++ podataka ili funkcija treba omogućiti pomoću *QObject* izvedene klase. Integracija s meta-objekt sustavom omogućuje QML-u pristupanje metodama, svojstvima i signalima bilo koje klase koja je izvedena iz *QObject* klase. Vidljivost QML-u može se ostvariti na više načina [5]:

- klasa može biti registrirana kao instancirani QML tip,
- registracija klase sa samo jednom instancom (engl. *Singleton*),
- instanca klase može biti ugrađena u QML kod.

To su najčešće metode koje omogućuju QML-u da pristupa C++ funkcionalnostima. Također je moguća i obrnuta interakcija tako da C++ upravlja QML objektima.

Izlaganje C++ atributa QML-u

Izlaganje C++ svojstava omogućuje komunikaciju između GUI dijela i dijela zaduženog za skladištenje i obradu podataka. Kao što je već rečeno, najznačajniju ulogu ima meta-objekt sustav koji to omogućuje. Omogućen je pristup sljedećim značajkama [5]:

- svojstva,
- metode (moraju biti označene kao javni slot ili im treba biti pridružena makronaredba Q_INVOKABLE kod deklaracije),
- signali.

Dodatno, dostupne su i enumeracije (engl. *enums*) ako su deklarirane s makronaredbom Q_ENUMS.

Izlaganje svojstava

Svojstvo može biti definirano u bilo kojoj klasi koja je izvedena iz *QObject* klase koristeći Q_PROPERTY() makronaredbu. Svojstvo je podatkovni član klase s pridruženim funkcijama za čitanje i pisanje. U nastavku se nalazi primjer klase *Message* koja za svojstvo ima varijablu *author*. Primjer je priložen pod Prilog 3.1 [5]. Svojstvo se može čitati pomoću metode *author*(), a postavljati, odnosno mijenjati pomoću metode *setAuthor*().
Primjer deklaracije Q_PROPERTY() :

```
Q_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)
```

Svako svojstvo koje može mijenjati vrijednost treba imati NOTIFY signal koji se odašilje prilikom svake promjene vrijednosti. Dakle, prilikom promjene svojstva, promjena je vidljiva i u GUI dijelu. U primjeru se vrijednost svojstva *author* mijenja u *Message::setAuthor()* i nakon toga se odašilje signal koji daje do znanja QML-u da se vrijednost *author* promijenila i ponovo

se poziva *Message::autor()* koji u GUI vraća varijablu *author* s novom vrijednosti. U slučaju izostanka NOTIFY signala, vrijednost bi bila postavljena na inicijalnu vrijednost koju vraća *Message:author()* ali u slučaju promjene iste, osvježenje GUI-a se neće dogoditi te će se pojaviti upozorenje iz QML dijela. Ako se varijabla ne mijenja tokom izvršenja aplikacije umjesto NOTIFY atributa potrebno je postaviti atribut CONSTANT.

Izlaganje metoda i slotova

Metoda iz *QObject* izvedene klase dostupna je QML-u ako je:

- javna metoda označena s makronaredbom Q_INVOKABLE(),
- metoda koja je javni Qt slot.

U prilogu 3.2. vidljivo je korištenje Q_INVOKABLE metode te Qt slota [5]. QML dio ima aktivan *MouseArea* na koji se klikom prvo poziva *MessageBoard::postMessage* i šalje poruku "Pozdrav iz QML-a" koja se pritom ispisuje u C++ dijelu te ista funkcija vraća *bool* vrijednost *true* koja postavlja QML varijablu *var result* na *true* i potom ispisuje poruku "Rezultat iz postMessage(): true". Na kraju se iz QML-a poziva slot u C++ dijelu naredbom *msgBoard.refresh().*

Izlaganje signala

Svaki javni signal *QObject* izvedene klase dostupan je QML-u. QML automatski stvara upravitelje signala (engl. *signal handler*). Upravitelji signala su uvijek nazvani kao "on<Signal>" gdje je <Signal> ime signala iz C++ dijela.

Primjer signala i upravitelja signala:

C++ kod:

```
class MessageBoard : public QObject
{
    Q_OBJECT
public:
   // ...
signals:
   void newMessagePosted(const QString &subject);
};
```

QML kod:

```
MessageBoard {
    onNewMessagePosted: console.log("Poruka primljena:", subject)
}
```

Važno je da su tipovi varijabli koje se šalju također podržani u QML-u. Konverzija između tipova opisana je u nastavku.

Konverzija tipova podataka između C++ i QML

Prilikom razmjene podataka između C++ dijela i QML-a, QML konvertira tipove u sebi čitljive. Postoje tipovi koje QML automatski konvertira i oni su vidljivi u tablici 3.1, a tipove koji nisu podržani potrebno je registrirati u C++ dijelu pozivom funkcije *qmlRegisterType().*

Tab. 3.1. *Konverzija tipova* [5]

Qt tip	**QML tip**
bool	bool
unsigned int, int	int
double	double
float, qreal	real
QString	string
QUrl	url
QColor	color
QFont	font
QDate	date
QPoint, QPointF	point
QSize, QSizeF	size
QRect, QRectF	rect
QMatrix4x4	matrix4x4
QQuaternion	quaternion
QVector2D, QVector3D, QVector4D	vector2d, vector3d, vector4d
Q_ENUMS()	enumeracija

3.1.5. Meta-objektni sustav

Meta-objektni sustav temelji se na tri stvari [5]:

- *QObject* klasa je baza za objekte koji mogu koristiti meta-objektni sustav,
- prilikom deklaracije klase potrebno je ubaciti Q_OBJECT makronaredbu unutar privatnog dijela kako bi klasi bilo omogućeno korištenje prednosti meta-objektnog sustava,
- meta-objektni prevoditelj (engl. *Meta-Object Compiler* – MOC) svakoj *QObject* podklasi dodjeljuje potreban kod kako bi bilo omogućeno korištenje meta-objekt značajki.

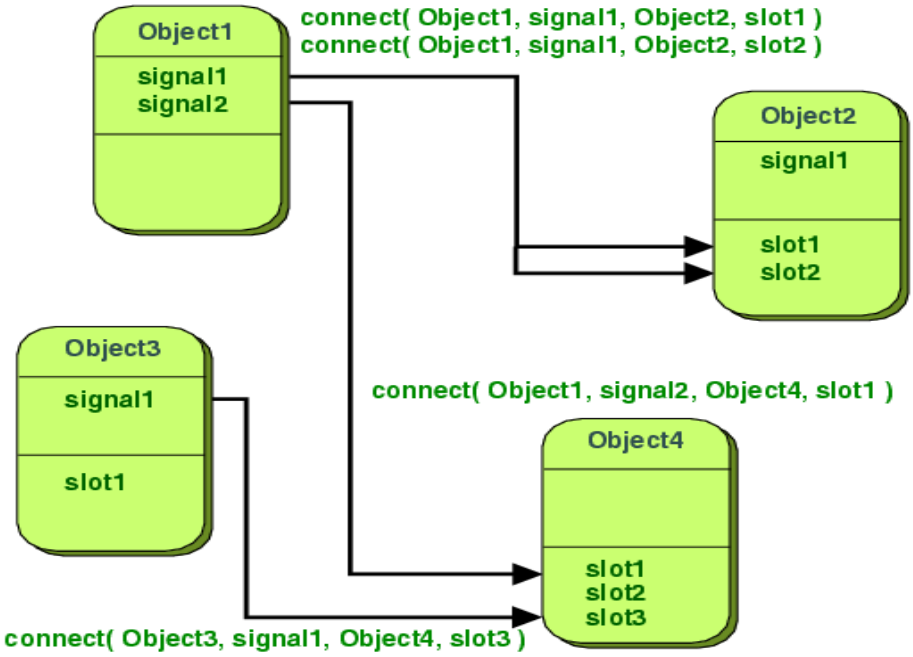
MOC alat čita C++ izvornu datoteku i ako se utvrdi da postoji jedna ili više klasa koje sadrže Q_OBJECT makronaredbu, stvara se još jedna C++ datoteka koja sadrži meta-objektni kod za svaku od klasa.

3.1.6. Mehanizam signala i slotova

Mehanizam signala i slotova jedan je od glavnih značajki Qt-a. Koristi se za komunikaciju između objekata, a omogućuje ga meta-objektni sustav. U GUI programiranju većina radnih okvira koristi *callback* funkcije koje obavljaju ovu vrstu komunikacije. *Callback* funkcije nisu uvijek sigurne i ne može se sa sigurnošću reći da će se pozvati *callback* s točnim argumentima te mu je mana što je ovisan o funkciji koja se izvodi. Kod usporedbe brzine izvođenja, mehanizam signala i slotova nešto je sporiji od *callback* funkcija zbog fleksibilnosti koju pruža.

Kod mehanizma signala i slotova kada se dogodi određeni događaj, odašilje se signal. Na odaslani signal aktivira se slot koji je zapravo funkcija koja je pozvana kao odgovor na određeni signal. Qt ima predefinirane signale i slotove, no isto tako je moguće proširiti klasu s vlastitim signalima i slotovima. Signali i slotovi mogu imati neodređen broj argumenata, a klasa koja odašilje signal ne mora znati koji će se slot aktivirati. Sve klase koje nasljeđuju *QObject* klasu ili neku njegovu podklasu mogu imati signale i slotove. Slotovi uz primanje signala, odnosno aktiviranje na signal, imaju ulogu i kao normalne funkcije. Isto kao što objekt ne zna da li je neki slot aktiviran ako odašilje signal, tako i slot ne zna da li je neki signal spojen na njega. Na taj način omogućeno je kreiranje neovisnih komponenti.

Na slici 3.3. je prikazano kako su spojeni signali i slotovi od četiri objekta. Vidljivo je da je signal moguće spojiti za više slotova; *signal1* objekta *Object1* spojen je za *slot1* objekta *Object2* te za *slot2* objekta *Object2*.



Sl. 3.3. *Spajanje signala i slotova* [5]

U ovom radu korišten je mehanizam signala i slotova između samih objekata u pozadinskom dijelu aplikacije (engl. *backend*) te između sučelja (engl. *frontend*) i pozadinskog dijela aplikacije. Razlika je jedino u tome kako se povezuju signali i slotovi.

Signali

Signali se odašilju kada se promijeni unutarnje stanje objekta, a promjena može utjecati na ostale objekte. To su funkcije javnog pristupa, no preporuka je da ga emitiraju samo klase gdje je signal definiran ili podklase. Slot na koji je signal spojen se izvršava odmah nakon odašiljanja signala. Ako je signal spojen na nekoliko slotova, izvršenje slotova će biti slijedno, redom kojim su spojeni. Signali ne smiju imati povratnu vrijednost (tip *void*) i automatski su generirani od strane MOC prevoditelja.

Slotovi

Slotovi su zapravo normalne C++ funkcije na koje se mogu spojiti signali. Funkcije kao slotove može pozvati bilo koja komponenta preko signal-slot veze, odnosno signal odaslan iz instance proizvoljne klase može pozvati slot instance neke nevezane klase.

3.1.7. Model/Pogled programiranje

Qt koristi skup klasa za pregled elemenata koje koriste model/pogled (engl. *model/view*) arhitekturu te tako omogućuje upravljanje odnosa između spremanja podataka i načina na koji su

podaci prikazani korisniku. Ta funkcionalnost omogućava programerima veću fleksibilnost prikazivanja elemenata i pruža sučelje modela koje omogućuje širok raspon izvora podataka koji će se koristiti s postojećim pregledima elemenata.

MVC (engl. *Model – View – Controller*) arhitektura koristi se u softverskom inženjeringu za odvajanje pojedinih dijelova aplikacije. Sastoji se od tri cjeline [5]:

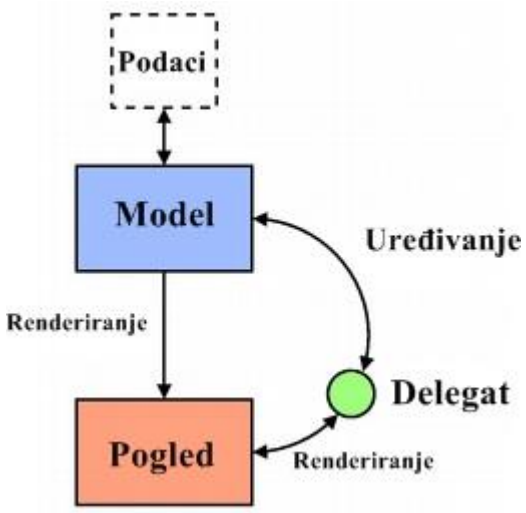
→ Model (engl. *Model*) : podaci i logika

→ Pogled (engl. *View*) : prikaz modeliranih podataka

→ Upravitelj (engl. *Controller*) : upravlja korisničkim zahtjevima

Model se sastoji od podataka i programske logike te obavještava pridružene poglede i upravitelje kada se dogodi neka promjena. Pogledi tako prikazuju promijenjeno stanje modela, a upravitelj ima novi skup naredbi. Pogled zahtjeva podatke od modela kako bi isti mogao prikazati korisniku. Upravitelj šalje zahtjeve modelu kojima se ažurira stanje modela (npr. Izmjena dokumenta) te može slati zahtjeve pogledu koji mijenja prikaz modela (npr. Kretanje kroz dokument).

Spajanjem pogleda i upravitelja nastaje model/pogled arhitektura. Ona uz jednostavniji radni okvir i dalje odvaja način na koji su podaci spremljeni od načina na koji su podaci prikazani korisniku. Odvajanje pruža mogućnost prikazivanja istih podataka u nekoliko različitih pogleda te izradu novih vrsta pregleda bez promjene strukture podataka. Na slici 3.4. prikazana je model/pogled arhitektura.



Sl. 3.4. *Model/Pogled arhitektura* [5]

Model komunicira s izvorom informacija i pruža sučelje drugim komponentama u arhitekturi (pogled, delegat). Način na koji model dohvaća informacije ovisi o tipu izvora te načinu implementacije modela. Pogled od modela dobiva indekse koji su reference na elemente

podataka. Ta informacija omogućava pogledu da dohvati elemente podataka iz izvora. Delegat prikazuje elemente podataka i komunicira s modelom koristeći indekse modela.

Modeli, pregledi i delegati međusobno komuniciraju pomoću mehanizma signala i slotova. Promjene u izvoru podataka mijenjaju podatke u modelu koji pritom odašilje signal koji obavještava pregled o promjenama. Signali iz pregleda pružaju informacije o interakciji korisnika s aplikacijom dok signali iz delegata služe prilikom uređivanja kako bi se obavijestili model i pregled o trenutnom stanju uređivanja.

Modeli

Svi modeli temelje se na *QAbstractItemModel* klasi. Ona definira sučelje koje koriste pogledi i delegati za pristup podacima. Klasa se ne instancira direktno, nego se kreiraju podklase za kreiranje modela. Za prikazivanje QML listi potrebno je koristiti *QAbstractListModel* koji nasljeđuje klasu *QAbstractItemModel*. *QAbstractListModel* klasa služi za kreiranje apstraktnih modela s jednodimenzionalnom listom što je upravo i korišteno prilikom izrade aplikacije. Za realizaciju EPG aplikacije s mogućnošću osvježavanja podataka potrebno je implementirati sljedeće funkcije [5]:

- *rowCount()*

Vraća broj djece od roditelja, odnosno broj redaka danog roditelja.

- *data()*

Sprema podatke pod određenom ulogom (engl. *role*) za element određen indeksom.

- *roleNames()*

Vraća imena uloga modela.

- *insertRows()*

Prilikom implementiranja potrebno je pozvati i određene funkcije kako bi korisniku automatski bile vidljive promjene. Prije ubacivanja podataka potrebno je pozvati funkciju *beginInsertRows(),* a nakon ubacivanja *endInsertRows().*

- *removeRows()*

Analogno ubacivanju, prije uklanjanja je potrebno pozvati *beginRemoveRows()*, a nakon uklanjanja *endRemoveRows()*.

Qt također pruža i gotove modele za korištenje kao što su:

- *QStringListModel* – pohrana jednostavne liste *QString* elemenata,
- *QStandardItemModel* – složeno stablo elemenata,
- *QFileSystemModel* – informacije o datotekama i direktorijima,

- *QSqlQueryModel*, *QSqlTableModel* i *QSqlRelationTableModel* – pristup bazama podataka.

Qt pruža brojne modele te funkcionalnosti, a detaljnije o svemu je moguće pronaći u dokumentaciji na službenoj stranici [4]. Pogledi i upravitelji realizirani su pomoću QML komponenti.

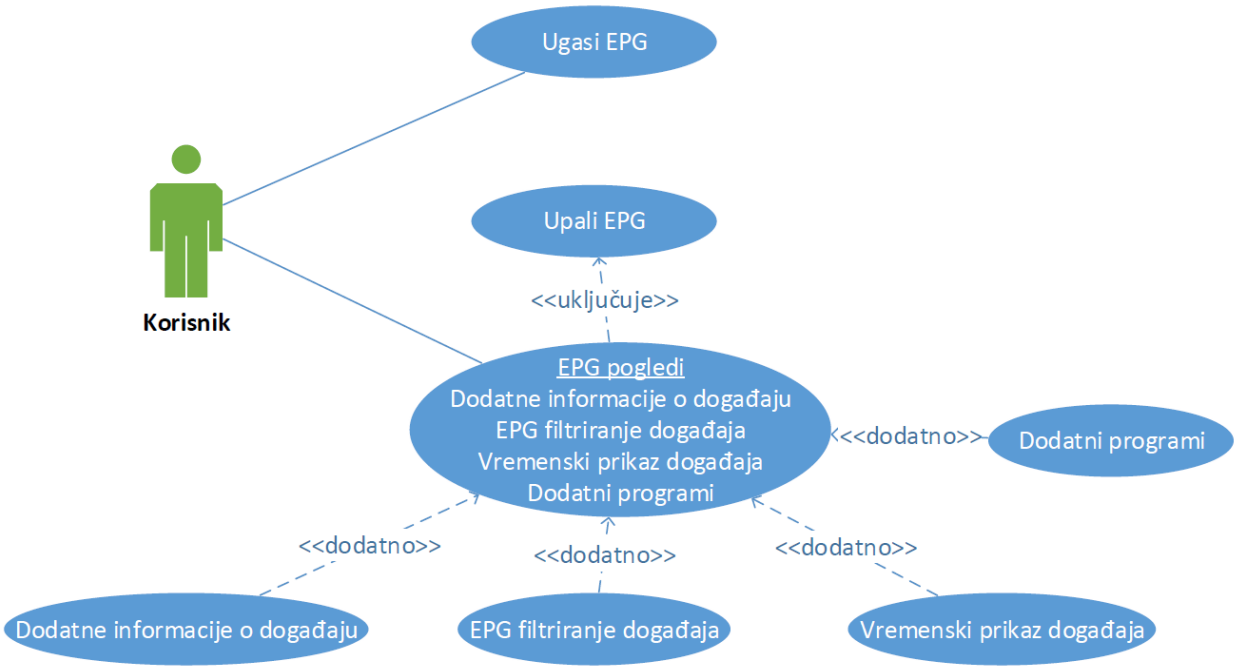
4. REALIZACIJA SUČELJA ZA ELEKTRONSKI PROGRAMSKI VODIČ

4.1. Koncept rješenja

Prije samog implementiranja aplikacije bilo je potrebno utvrditi način na koji će se realizirati pojedini moduli i na koji način će komunicirati, odnosno izmjenjivati poruke i podatke. Također je vrlo bitno na koji način će gledatelji koristiti aplikaciju te su stoga formirani dijagram slučajeva i sekvencijalni dijagram. Isti su prikazani u nastavku.

4.1.1. Dijagram slučajeva

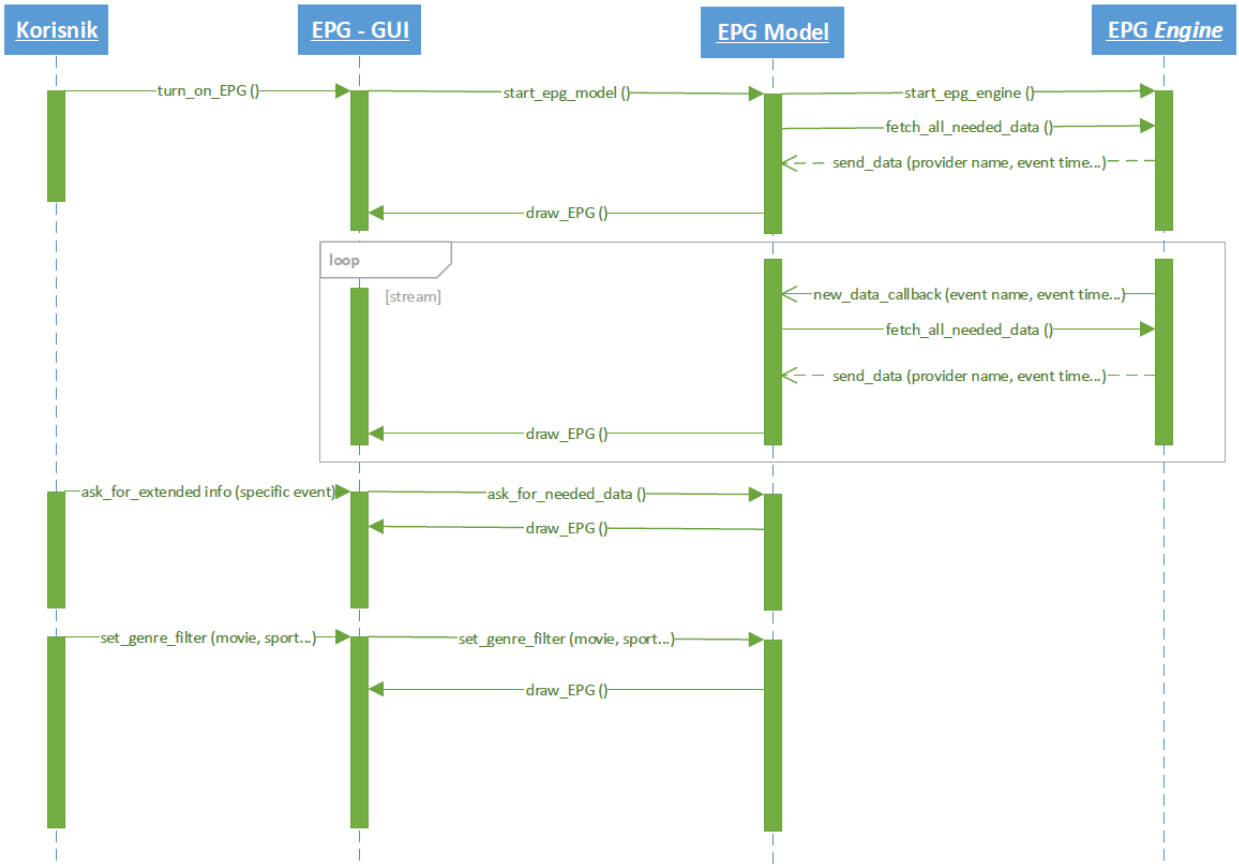
Dijagram slučajeva ključan je s gledateljeva gledišta, što je vidljivo na slici 4.1. Kada korisnik upali aplikaciju automatski mu je vidljiv jedan od dva prikaza EPG-a (prikaz koji prikazuje informacije o trenutnom i nadolazećem događaju određenog kanala te prikaz koji prikazuje sve događaje kanala u određenom vremenskom razdoblju). I u jednom i u drugom prikazu korisnik se može kretati kroz događaje te vidjeti dodatne informacije o događaju. Također se može kretati kroz listu kanala kako bi vidio i ostale kanale koji trenutno nisu vidljivi, odnosno nisu u fokusu. Kada je upaljen vremenski prikaz događaja korisnik može filtrirati događaje prema ponuđenim žanrovima.



Sl. 4.1. *Dijagram slučajeva za korištenje EPG aplikacije*

4.1.2. Sekvencijalni dijagram

Sekvencijalni dijagram ključan je za programera kako bi si lakše predočio samu implementaciju aplikacije i podijelio jedan veliki problem na više manjih problema. Dijagram prikazuje otprilike na koji će se način implementirati aplikacija i prikazan je na slici 4.2.



Sl. 4.2. *Sekvencijalni dijagram rada EPG aplikacije*

4.2. Definicija korisničkog sučelja

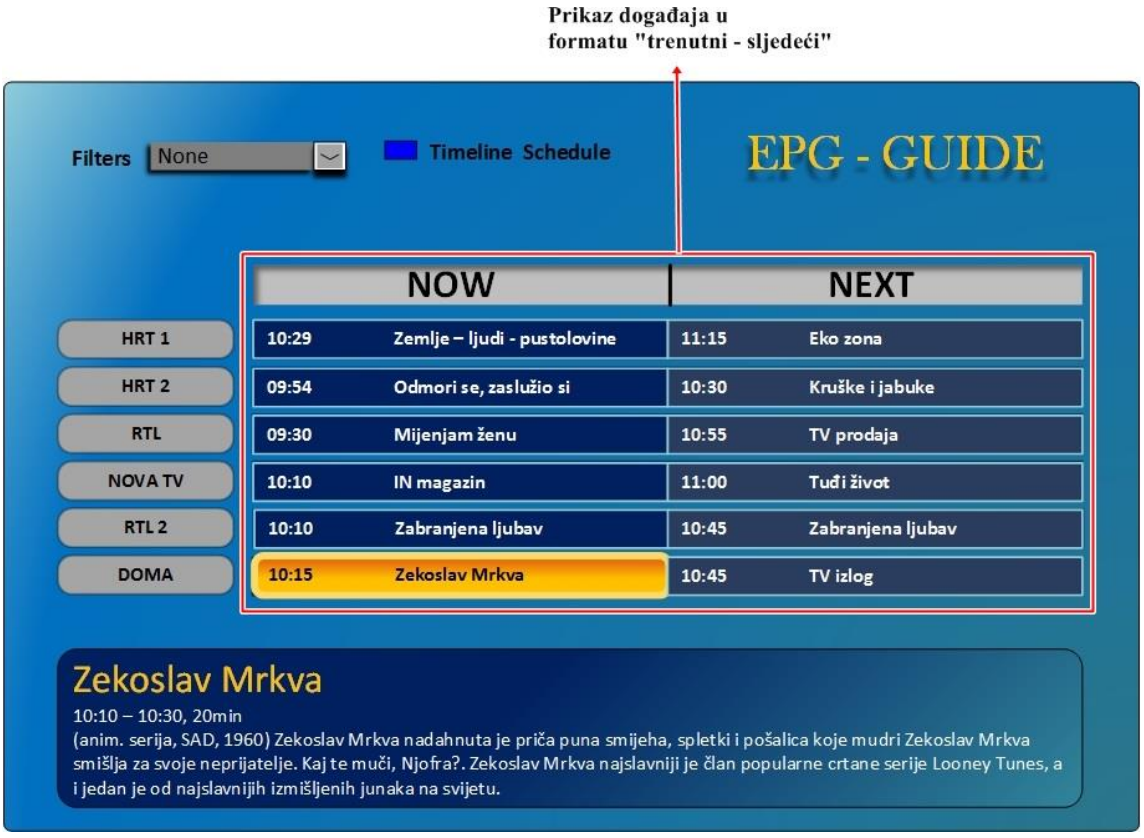
Uz dijagrame je također napravljen dizajn kako bi aplikacija trebala izgledati. Na slici 4.3. prikazan je izgled aplikacije te su na njoj objašnjeni sastavni dijelovi sučelja. Prikazano je kako aplikacija izgleda u trenutku kada je aktivan vremenski prikaz događaja te nije uključeno filtriranje događaja.

Sl. 4.3. *Dizajn EPG aplikacije – vremenski prikaz događaja*

Na slici 4.4. prikazano je kako aplikacija izgleda kada je odabran filter, odnosno žanr događaja. Svi događaji koji ne pripadaju pod odabrani žanr su zasjenjeni, dok je boja događaja koji odgovaraju žanru nepromijenjena.

Sl. 4.4. *Dizajn EPG aplikacije – vremenski prikaz događaja s odabranim filterom*

Slika 4.5. prikazuje izgled aplikacije kada je odabran prikaz trenutnog i sljedećeg događaja pojedinog kanala.



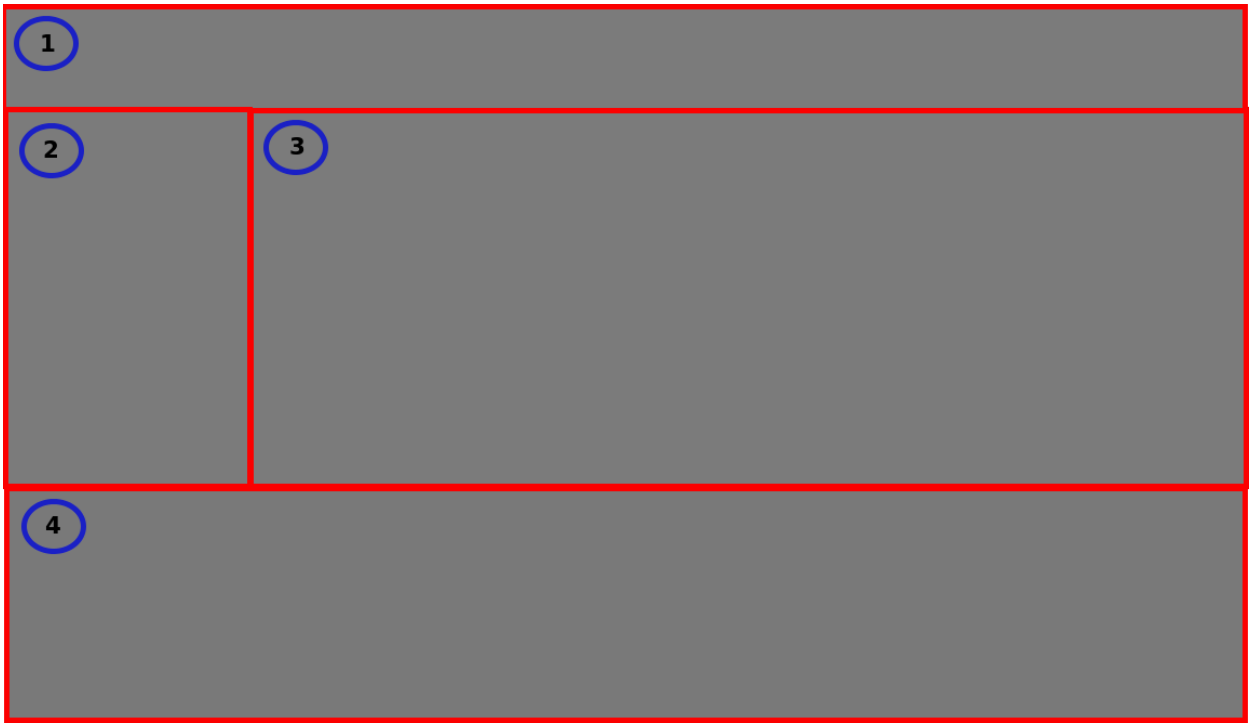
Sl. 4.5. *Dizajn EPG aplikacije – trenutni i sljedeći događaj*

4.3. Moduli

Aplikacija se sastoji od više modula koji sadrže više datoteka, stoga su u nastavku opisana tri glavna modula: EPG *View*, EPG Model, EPG *Engine*.

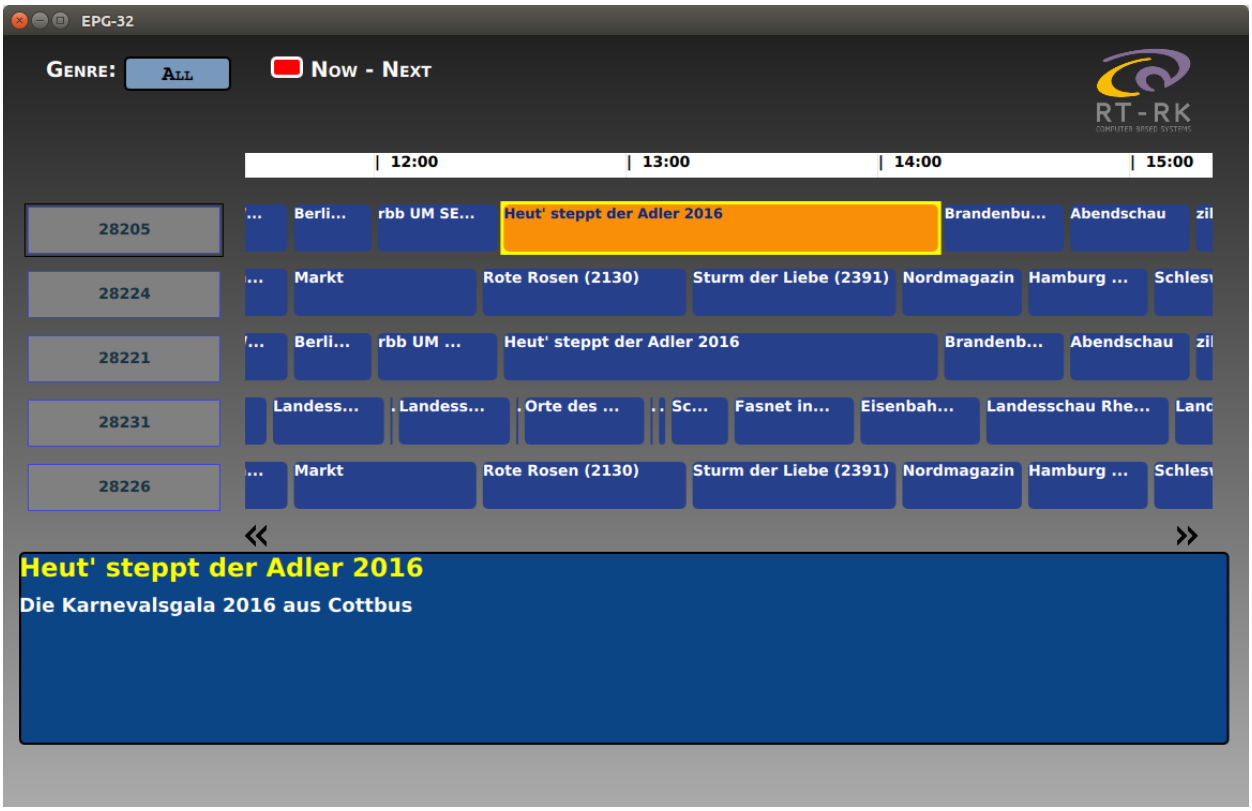
4.3.1. EPG View

EPG *View* je zapravo GUI dio te je ujedno najzanimljiviji korisniku. Realiziran je pomoću QML-a te JavaScript programskog jezika. QML je zaslužan za vizualne efekte, dok JavaScript omogućava dohvaćanje podataka te interakciju između elemenata formiranih QML-om. GUI se sastoji od glavnog prozora koji se može podijeliti na 4 dijela prikazana na slici 4.6. Dijelovi 1, 2 i 4 su uvijek prisutni dok dio 3 ima dvije opcije prikazivanja.

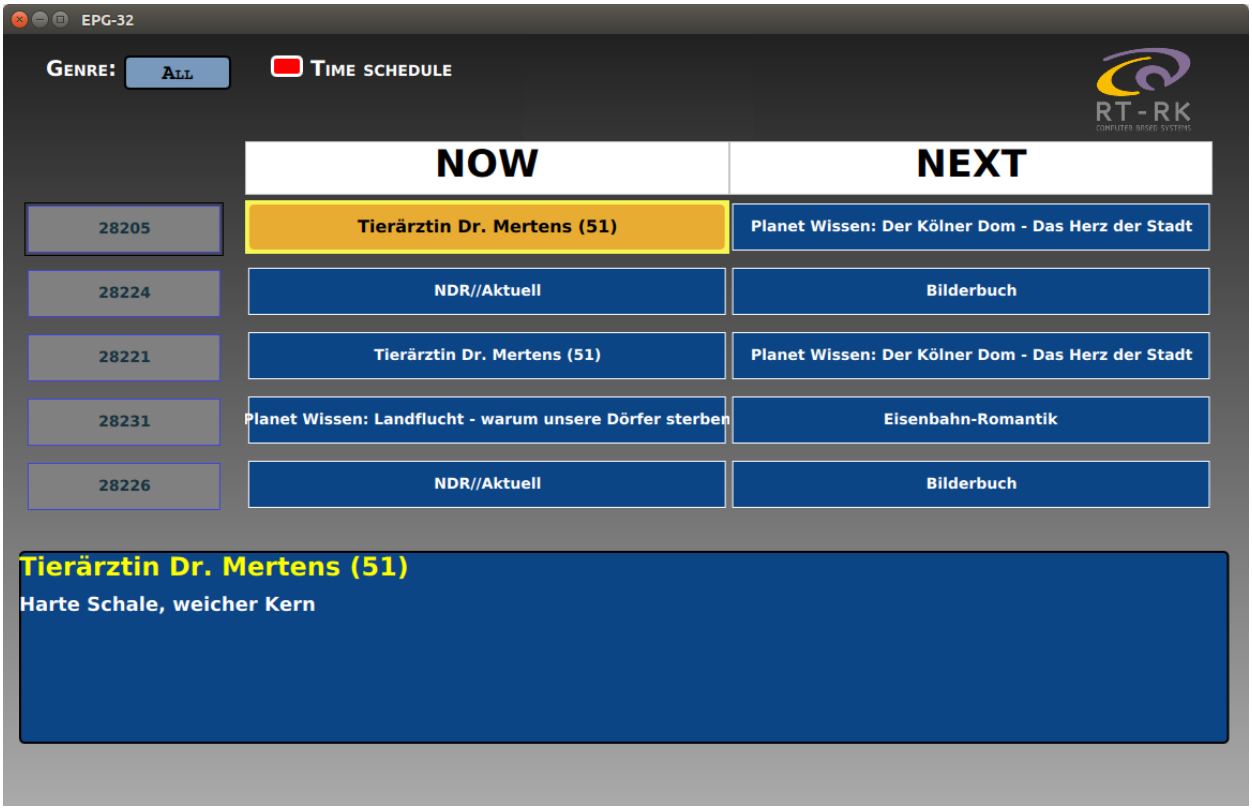


Sl. 4.6. *Struktura sučelja EPG aplikacije*

Na slici 4.7. prikazan je vremenski prikaz događaja, a na slici 4.8. trenutno-sljedeći prikaz događaja.



Sl. 4.7. *Konačan izgled EPG aplikacije – vremenski prikaz*



Sl. 4.8. *Konačan izgled EPG aplikacije – trenutno-sljedeći*

Zaglavlje

Prvi dio sa slike 4.6. predstavlja zaglavlje (Slika 4.9.). Zaglavlje se sastoji od padajućeg izbornika gdje je moguće odabrati koji žanr prikazati i tipke (engl. *Button*) koja omogućuje mijenjanje između vremenskog prikaza događaja te opcije trenutno-sljedeći prikaz događaja.

Sl. 4.9. *Zaglavlje EPG View modula*

Prilikom pokretanja tekst pored tipke je "Time schedule" budući da je na početku aktivan trenutno-sljedeći način prikazivanja događaja. Kada se uključi vremenski prikaz događaja "Time schedule" tekst se mijenja u "Now – Next". Uz mijenjanje između načina prikazivanja događaja, tipka ima još nekoliko funkcija, a to su:

- Prilikom uključivanja vremenskog prikaza događaja, uključuje se vidljivost strelica u donjem dijelu ekrana koje služe kao pokazatelj korisniku da postoji više događaja u vremenskoj crti. U suprotnom, kada se uključuje trenutno-sljedeći prikaz događaja, strelice se isključuju.
- Prilikom izmjene načina prikazivanja fokus se automatski postavlja na početak liste kanala.

Lista s kanalima

Ispod zaglavlja se nalazi lista s kanalima te pravokutnik gdje se nalaze događaji. Od ukupne širine glavnog prozora lista s kanalima zauzima jednu petinu, dok pravokutnik gdje se nalaze događaji zauzima četiri petine prostora. Najvažniji gradivni element liste kanala te općenito EPG aplikacije je *ListView* koji prikazuje podatke modela koji je kreiran u QML-u kao *ListModel* i *XmlListModel* ili modela definiranog u C++ dijelu. Prilikom izrade aplikacije za testne slučajeve korišten je QML model te je kasnije kreiran C++ model koji se koristi u završnoj verziji aplikacije. Budući da *ListView* kao gradivni element ima veliki broj svojstava, jednostavan primjer korištenja elementa *ListView* dan je u prilogu 4.1**,** a u radu su opisana najbitnija i najviše korištena svojstva za pravilan rad aplikacije.

Ukupno je korišteno osam listi za prikaz podataka:

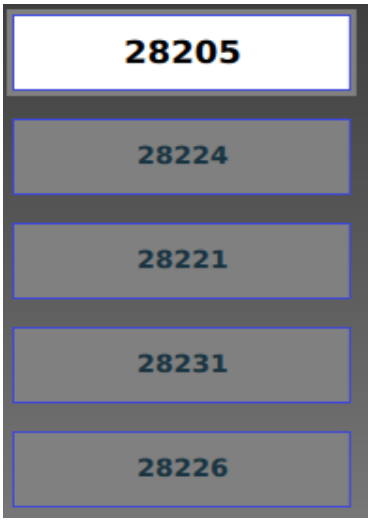
1. Lista kanala – identifikator: *serviceList*
2. Lista za prikaz trenutnih događaja – identifikator: *eventListPresent*

3. Lista za prikaz nadolazećih događaja – identifikator: *eventListFollowing*

4. Pet listi za vremenski prikaz događaja – *eventListHorizontal1 – eventListHorizontal5*

Korisniku je odjednom vidljivo maksimalno pet kanala te ako želi vidjeti ostale kanale potrebno je navigirati ili kroz listu kanala ili kroz listu događaja.

Kod pokretanja programa te izmjene načina prikazivanja događaja fokus je uvijek na listi kanala, a prilikom navigacije fokus se mijenja. Jedan je kanal istaknut dok su ostali kanali prikazani normalno (Slika 4.10.).



Sl. 4.10. *Lista s kanalima*

Iz liste kanala pritiskom tipke "strelica desno" omogućen je prijelaz u listu za prikaz trenutnih događaja ili u jednu od horizontalnih lista. Primjer:

```
KeyNavigation.right: eventListPresent.visible ? eventListPresent : eventListHorizontal1
```

Kod navigacije gore-dolje, odnosno pomicanja, promjena trenutno prikazanih elemenata mijenja svojstvo *contentY* (koordinate liste). Npr. na početku je prikazano prvih pet kanala i tada *contentY* iznosi 0, a kada korisnik navigiranjem dođe do šestog elementa u listi, *contentY* se mijenja i tada iznosi upravo onoliko kolika je visina jednog elementa (u aplikaciji iznosi 60 elemenata slike). Prilikom promjene, poziva se funkcija *scrollAtTheSameTime (listId)* koja kao argument prima identifikator liste iz koje je pozvana. Kao što sam naziv funkcije kaže, ona omogućava pomicanje svih listi istovremeno. To je izvedeno na način da se svojstvo *contentY* mijenja ostalim listama na vrijednost liste iz koje je funkcija pozvana. Iz liste kanala je također moguće mijenjati modele listi *eventListHorizontal1 – eventListHorizontal5*, no samo ako je u tom trenutku aktivan vremenski prikaz događaja.

Primjer:

```
function scrollAtTheSameTime (listId)
   {
      if (listId === serviceList)
      {
         eventListPresent.contentY = serviceList.contentY
         eventListFollowing.contentY = serviceList.contentY
      }
      else if (listId === eventListPresent)
      {
         serviceList.contentY = eventListPresent.contentY
         eventListFollowing.contentY = eventListPresent.contentY
      }
      else if (listId === eventListFollowing)
      {
         serviceList.contentY = eventListFollowing.contentY
         eventListPresent.contentY = eventListFollowing.contentY
       }
   }
```

Pravokutnik za prikaz događaja

Treći dio sa slike 4.6. služi za prikaz događaja od aktivnih kanala. Moguć je prikaz trenutno-sljedeći i vremenski prikaz događaja. Način prikaza se mijenja na već opisani način pomoću tipke u zaglavlju, a prilikom pokretanja je aktivan trenutno-sljedeći način prikazivanja (Slika 4.7.) .

Trenutno-sljedeći prikaz događaja

Trenutno-sljedeći prikaz se sastoji od dvije tekstualne oznake koje se nalaze na vrhu iznad svake liste – oznaka "NOW" i oznaka "NEXT". Lista za prikaz trenutnih događaja i lista za prikaz nadolazećih događaja imaju identična svojstva, razlikuju se samo po modelima koji su im pridruženi iz C++ dijela. Iz liste za trenutni prikaz moguće je navigirati do liste kanala i do liste za prikaz nadolazećih događaja, dok je iz liste za prikaz nadolazećih događaja moguće navigirati samo do liste za prikaz trenutnih događaja. Iz obje liste se prilikom promjene *contentY* svojstva poziva opisana funkcija *scrollAtTheSameTime (listId)*.

Kako bi iz elementa jedne liste prešli u paralelni element druge liste, važno je prilikom prolaska kroz listu mijenjati trenutne indekse ostalih lista. Primjer prilikom promjene indeksa u *eventListPresent* mijenja se trenutni indeks *serviceList* i *eventListFollowing*:

```
onCurrentIndexChanged: {
                serviceList.currentIndex = eventListPresent.currentIndex
                eventListFollowing.currentIndex = eventListPresent.currentIndex
                getExtendedData("nowList", eventList.currentIndex+1)
            }
```

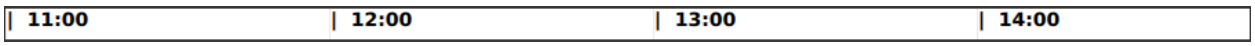
Također se odašilje signal *getExtendedData* koji je detaljnije opisan u dijelu 4.4.2.

Vremenski prikaz događaja

Vremenski prikaz događaja sastoji se od vremenske trake i listi događaja.

Vremenska traka

Vremenska traka prikazuje vrijeme u razmacima od jednog sata (Slika 4.11.). Traka se nalazi na mjestu gdje se inače nalaze tekstualne oznake "NOW" i "NEXT" kada je aktivan trenutno-sljedeći prikaz događaja. Traka je samo za vizualni prikaz te istom nije moguće navigirati niti ju označavati (svojstvo → *interactive: false*).



Sl.4.11. *Vremenska traka za označavanje sati u vremenskom prikazu događaja*

Liste događaja

Ispod vremenske trake nalazi se 5 lista (*eventListHorizontal1 – eventListHorizontal5*) koje prikazuju događaje pet trenutno prikazanih kanala. Vremenska traka i svih pet lista su horizontalno orijentirane (svojstvo → *orientation: ListView.Horizontal*). Širina vidljivog dijela liste pokriva ukupno četiri sata vremenskog prikaza.

Navigacija kroz liste

Navigacija kroz listu je ugrađeno svojstvo, no da bi se navigiralo između lista potrebno je pozvati funkciju koja aktivira fokus. Vertikalni prijelaz između listi riješen je s nekoliko funckija. Npr. prijelaz iz prve u drugu listu:

```
Keys.onDownPressed: {

                horizontal2.forceActiveFocus()
                mySignal(modelListThree, (horizontal2.currentIndex + 1))
            }
```

Iz druge liste je moguće prijeći u prvu i treću te je potrebno dodati sljedeće:

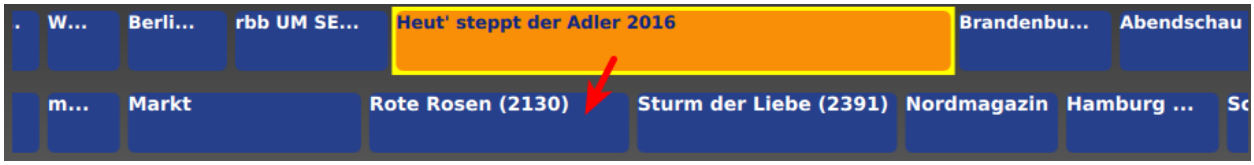
```
Keys.onUpPressed: {
                horizontal.forceActiveFocus()
                mySignal(modelListOne, (horizontal.currentIndex + 1))
            }
Keys.onDownPressed: {
                horizontal3.forceActiveFocus()
                mySignal(modelListThree, (horizontal3.currentIndex + 1))
            }
```

Analogno tome, problem je riješen za ostale liste.

Horizontalno pomicanje po elementima liste je ugrađeno svojstvo QML listi. Prilikom pomicanja kroz liste mijenja se trenutni indeks liste koja je trenutno u fokusu. Promjenom indeksa, odnosno navigiranja kroz elemente potrebno je i ostalim listama promijeniti indeks kako bi se prilikom navigiranja između listi fokus zadržao na sredini vidljivih dijelova liste. Budući da se zna širina listi (940 elemenata slike) traži se indeks liste koji se nalazi na 470 elemenata slike. Funkcija *int indexAt(int x, int y)* vraća indeks liste koji se nalazi na traženim koordinatama x i y, u suprotnom ako indeks nije pronađen funkcija vraća vrijednost -1. Trenutni indeks se mijenja na sljedeći način:

```
eventListHorizontal1.currentIndex=
eventListHorizontal1.indexAt((eventListHorizontal1.contentX + 470), 0)
```

Analogno tome, indeks se mijenja ostalim listama. X vrijednost se uvijek postavlja na vrijednost *horizontal.contentX* + 470. Budući da *contentX* označava koliko je lista pomaknuta po x-osi potrebno je zbrojiti *contentX* sa 470 što je na polovici vidljivog dijela liste. Na slici 4.12. prikazano je koji će se događaj aktivirati prilikom navigacije između dvije liste. Oba događaja ispod trenutno aktivnog događaja su na sredini, no fokus će se aktivirati na događaju koji je na poziciji lijevo ispod.



Sl. 4.12. *Dio za navigaciju između događaja*

Prijelazom na element koji nije u vidljivom dijelu liste, lista se pomjera lijevo/desno kako bi element postao vidljiv. Tim postupkom mijenja se svojstvo *contentX* koje ima istu ulogu kao

contentY kod vertikalnih listi – istovremeno pomicanje svih listi. Ako se jednoj listi promijeni vrijednost *contentX*, automatski se poziva mijenjanje *contentX* ostalim listama. Također se omogućuje vidljivost strelica koje pokazuju da li je dostupno još događaja – lijevo ili desno.

Primjer:

```
onContentXChanged: {
                if (eventListHorizontal1.activeFocus === true)
                {
                   eventListHorizontal2.contentX = eventListHorizontal1.contentX
                   eventListHorizontal3.contentX = eventListHorizontal1.contentX
                   eventListHorizontal4.contentX = eventListHorizontal1.contentX
                   eventListHorizontal5.contentX = eventListHorizontal1.contentX
                   timelinelista.contentX = eventListHorizontal1.contentX
                   if (eventListHorizontal1.contentX >= pragZaUcitavanje)
                   {
                      console.log(eventListHorizontal1.contentX)
                      pragZaUcitavanje+=940
                      addNewEventsToDatabase()
                   }
                   else if (eventListHorizontal1.contentX > 0)
                   {
                      arrowLeft.visible = true
                   }
                   else if (eventListHorizontal1.contentX === 0)
                   {
                      arrowLeft.visible = false
                   }
                }
```

Promjena modela prilikom navigiranja

Svakoj listi dodijeljen je određeni model koji je registriran u C++ dijelu aplikacije. Postoji niz sa registriranim modelima, a na temelju indeksa se radi izmjena. Na početku su listama dodijeljeni modeli koji se nalaze na indeksima od nula do četiri. Ako je korisnik na petom kanalu i pritisne strelicu prema dolje tada se poziva funkcija za izmjenu modela ukoliko postoji više od pet modela. Indeks modela svake liste se tada povećava za jedan i radi se izmjena modela – u tom trenutku aktivni su modeli od jedan do pet. Analogno tome, izmjena se radi i za pomicanje prema gore kada se korisnik nalazi na prvoj listi.

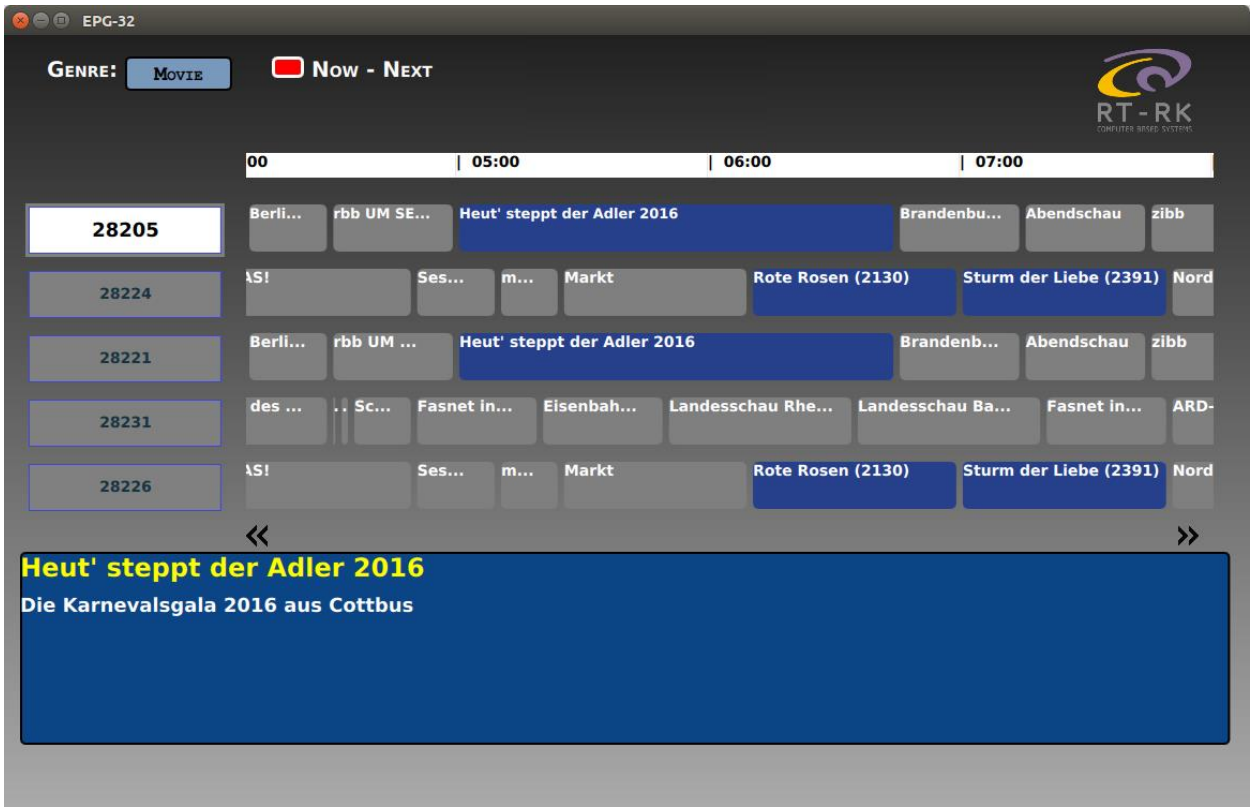
Prikazivanje pojedinog događaja

Svaki element liste prikazuje ime događaja, a širina svakog elementa je ovisna o trajanju događaja. Sva svojstva su proslijeđena iz C++ dijela. Boja svakog elementa se mijenja u ovisnosti o odabiru žanra, a ako je odabrana opcija "All" tada su svi elementi jednake boje. Količina teksta koja je prikazana ovisi o tome da li je tekst širi od pravokutnika elementa. U slučaju da jest, tekst se smanjuje za jedan znak dok širina teksta ne postane manja od širine pravokutnika i na kraju se još oduzimaju tri znaka kako bi se dodao znak "…". Primjer:

```
Component.onCompleted: {
            if (labelHorizontal.paintedWidth > insideConHorizontal.width)
           {
                var characterReducer = 20
                while (labelHorizontal.paintedWidth > insideConHorizontal.width)
                {
            labelHorizontal.text=labelHorizontal.text.substring(0,characterReducer)
            characterReducer -=1
                }
            characterReducer -=3
            labelHorizontal.text = labelHorizontal.text.substring(0,characterReducer)
            labelHorizontal.text += '...'
          }
              if (insideConHorizontal.width < 30)
                {
                   labelHorizontal.text = "."
              }
       }
```

Prikazivanje filtriranih događaja

Na početku programa svi događaji prikazani su istom bojom budući da nije odabran nijedan filter, odnosno žanr. Odabirom žanra poziva se funkcija koja mijenja svojstvo boje pojedinog događaja. Poziv funkcije obavlja se iz padajuće liste koja se nalazi u zaglavlju, a na slici 4.13. prikazan je EPG kada je odabran žanr "Movies".



Sl. 4.13. *Vremenski prikaz događaja – filtrirani događaji*

Dodatni prikaz informacija

Posljednji, četvrti dio sa slike 4.6. predstavlja pravokutnik gdje se prikazuju dodatne informacije o događaju ukoliko su dostupne (Slika 4.14.). Sastoji se od pravokutnika i nekoliko elemenata za prikaz teksta. Podatke koje prikazuje pristižu iz C++ dijela, a mijenjaju se prilikom navigiranja kroz liste događaja.

Sl. 4.14. *Prikaz dodatnih informacija EPG događaja*

4.3.2. EPG Model

Srž aplikacije realizirana je pomoću C++ programskog jezika i klasa koje pruža Qt. Zbog načina na koji je implementiran EPG *View* najvažniji gradivni element su također liste koje se popunjavaju podacima koji se dobivaju od EPG *Engine* modula napisanog u C programskom jeziku. Popis korištenih klasa je sljedeći:

- *Service,*

- *ServiceModel,*
- *EventTimeline,*
- *EventModelTimeline,*
- *Event,*
- *EventModel,*
- *TimeLineItem,*
- *TimeLineTrack,*
- *Consumer,*
- *Producer,*
- *NewEventsThread,*
- *NewServicesThread.*

ServiceModel

Klasa *Service* sadrži ime kanala i identifikator kanala. *ServiceModel* klasa nasljeđuje klasu *QAbstractListModel* koja je opisana u dijelu 3.1.7. Sadrži listu koja je popunjena objektima klase *Service,* te funkcije koje su potrebne za popunjavanje liste te izmjenu iste. Također sadrži *hash* tablicu koja omogućava prikazivanje podataka u EPG *View* dijelu.

EventModelTimeline

EventModelTimeline od podatkovnog dijela sadrži identifikator kanala te listu koja sadrži objekte klase *EventTimeline*. Pruža podatke za opisane horizontalne liste u EPG *View* dijelu. Prilikom obrade konfiguracijske datoteke koja sadrži broj traženih kanala te informacije o istim, svakoj instanci se dodijeli identifikator kanala te se na taj način registrira kojoj instanci pripada događaj koji je stigao iz prijenosnog toka. *Hash* tablica koju koristi omogućava prikazivanje podataka iz liste, a korištene su ključne riječi za ime događaja, opis, trajanje, početak i kraj događaja, žanr te boja kojom je događaj prikazan. Korištene funkcije omogućavaju popunjavanje liste, izmjenu liste te izmjenu boje događaja kada je aktiviran filter.

EventModel

EventModel klasa slična je klasi *EventModelTimeline,* a koristi se za spremanje i manipuliranje podacima za liste koje prikazuju trenutne i nadolazeće događaje (trenutno-sljedeći prikaz događaja). Liste sadrže objekte klase *Event*.

TimeLineTrack

TimeLineTrack klasa sadrži listu koja pruža podatke za prikaz vremenske trake. Lista sadrži objekte klase *TimeLineItem*.

Consumer

Consumer klasa koristi funkcije koje popunjavaju sve modele koji su korišteni te pruža informacije EPG *View*-u. Funkcije su deklarirane kao slotovi budući da se ne pozivaju iz iste programske niti. Funkcije su sljedeće:

```
void addEventSchedule (…);
```

Funkcija se poziva kada je dohvaćen događaj tipa eEIT_TYPE_SCHEDULE te se prolazi kroz sve instance klase *EventModelTimeline* kako bi se pronašao identifikator kanala koji se podudara sa identifikatorom kanala od dohvaćenog događaja. Kada se identifikator kanala podudara, odnosno kada se identificira lista u koju se treba staviti događaj – poziva se funkcija za dodavanje događaja u listu.

```
void addEventNow (…);
```

Analogno funkciji za dodavanje događaja u liste koje prikazuju događaje tipa eEIT_TYPE_SCHEDULE, funkcija *addEventNow* dodaje događaje tipa eEIT_TYPE_PRESENT.

```
void addEventNext (…);
```

Funkcija ima istu ulogu kao i funkcija *addEventNow i* dodaje događaje tipa eEIT_TYPE_FOLLOWING u za to predodređeni model.

```
void addNewEventsToDatabase ();
```

Na zahtjev iz GUI-a, odnosno na zahtjev korisnika potrebno je učitati nove podatke koji su dostupni. Uz dostupne događaje koji postoje za tražene kanale, dodaju se i sati u vremensku traku.

```
void endOfFirstReading ();
```

Svaki puta nakon što se učitaju događaji za nove kanale poziva se funkcija *endOfFirstReading* koja popunjava sve modele za koje nisu dostupni podaci sa podacima gdje su ime i opis događaja označeni sa "*No data available*".

```
void timeSignal(uint16_t time);
```

Funkcija kao parametar prima trenutno vrijeme iz prijenosnog toka te se iz te funkcije odašilje signal u GUI gdje se vrijednost koristi u svrhu pozicioniranja listi s događajima.

```
void sendNumberOfModels (uint16_t modelNumber);
```

Funkcija kao parametar prima broj modela koji je dobiven parsiranjem konfiguracijske datoteke. Iz funkcije se odašilje signal s brojem modela u GUI dio.

```
void extendedDataRequest (QString list, QString indeks);
```

Funkcija se poziva iz GUI dijela prilikom svake promjene indeksa. Kao parametre prima broj liste modela koji je aktivan, a ukoliko je riječ o listama za trenutno-sljedeći prikaz tada se šalje identifikator liste. Iz funkcije se nazad u GUI dio odašilje signal sa zahtjevanim podacima.
Signali koji su korišteni su sljedeći:

- *void setExtendedData (QVariant name, QVariant extendedText),*
- *void setTime (QVariant time),*
- *void setNumberOfModels (QVariant modelNumber).*

Signali su opisani u dijelu 4.4.2.

Producer

Klasa *Producer* je zapravo programska nit koja se pokreće iz glavne funkcije programa. Sadrži jednu funkciju – *run()* koja predstavlja programsku nit te signale koji povezuju klasu *Producer* i klasu *Consumer*. U programskoj niti se postavljaju zahtjevi za dohvaćanje podataka prilikom pokretanja aplikacije. Postavljaju se zahtjevi za događaje tipa eEIT_TYPE_PRESENT, eEIT_TYPE_FOLLOWING i eEIT_TYPE_SCHEDULE. Također se dohvaćaju podaci za četiri sata unaprijed tako da podaci budu spremni kada korisnik postavi zahtjev.
Signali koji su korišteni su sljedeći:

- *void newEvent (QString name, QString description, float duration, uint16_t serviceId, uint8_t genreEvent),*
- *void newEventNow (QString name, QString description, uint16_t serviceId),*
- *void newEventNext (QString name, QString description, uint16_t serviceId),*
- *void timeSignal(uint16_t time),*
- *void sendNumberOfModels(uint16_t numberOfModels),*
- *void endOfFirstReading(uint8_t endFlag).*

NewEventsThread

Klasa *NewEventsThread* ima jednu funkciju – *run* (). To je programska nit koja dohvaća podatke za četiri sata ukoliko su podaci dostupni. Na početku je prikupljeno osam sati događaja te je spremno četiri sata događaja ukoliko korisnik to zatraži. Ako se prikaže i tih četiri sata, dakle ukupno 12 sati, postavlja se zastavica koja omogućuje dohvaćanje novih četiri sata događaja kako bi događaji odmah bili spremni za prikaz.

NewServicesThread

Klasa *NewServicesThread* također nasljeđuje klasu *QThread*. U programskoj niti se na temelju zahtjeva iz GUI-a postavljaju zahtjevi EPG *Engine*-u za dohvaćanje događaja kanala koji još nisu prikazani.

Signali koji su korišteni su sljedeći:

- *void newServiceEvent (QString name, QString description, float duration, uint16_t serviceId, uint8_t genreEvent),*
- void *endOfFirstReading(uint8_t endFlag).*

4.3.3. EPG Engine

EPG *Engine* modul napisan je u C programskom jeziku. Njegova zadaća je povezivanje sa podatkovnim tokom, dobavljanje EIT sekcija iz toka podataka, obrada podataka te punjenje baze. Povezan je s modulom EPG Model i prosljeđuje mu dostupne podatke.

Za pokretanje i zaustavljanje modula EPG *Engine* korištene su sljedeće dvije funkcije:

- *bool MW_Run(void),*
- *bool MW_Stop(void).*

Prethodno navedene funkcije su potrebne za inicijalizaciju i deinicijalizaciju tunera, demodulatora i demultipleksera, inicijalizaciju i deinicijalizaciju EPG *Engine* modula te postavljanje frekvencije.

Datoteka eit.h sadrži deklaracije funkcija kao što su registriranje klijenta, kreiranje zahtjeva, dohvaćanje događaja, postavljanje jezika, postavljanje zemlje, dohvaćanje dodatnih informacija o događaju.

Funkcije iz eit.h datoteke omogućavaju dohvaćanje obrađenih podataka:

```
tEIT_Status EIT_RegisterClient(tEIT_Notification function, tEIT_ClientId* clientId);
```

Registriranje klijenta te *callback* funkcije koja će se pozvati ukoliko dođe do promjene.

```
tEIT_Status EIT_CreateRequest(tEIT_ClientId clientId, tEIT_DVBTriplet* dvbTripletArray, uint16_t dvbTripletArrayLength, tEIT_EventType eventType, tEIT_UTCTime startTime, tEIT_UTCTime endTime, tEIT_Genre genre, tEIT_RequestId* requestId);
```

Kreiranje zahtjeva za dohvaćanje podataka. Funkcija kao parametre prima identifikator klijenta, *dvbTripletArray* koji predstavlja niz kanala, broj kanala, tip događaja, vrijeme početka i kraja, tip žanra te izlazni parametar *requestId*.
Za tip događaja je potrebno postaviti jednu od tri opcije: eEIT_TYPE_PRESENT, eEIT_TYPE_FOLLOWING ili eEIT_TYPE_SCHEDULE. Odabir žanra također ima više opcija, no u aplikaciji se uvijek dohvaćaju događaji svih tipova – eEIT_ALL_GENRES.

```
tEIT_Status EIT_ReleaseRequest(tEIT_ClientId clientId, tEIT_RequestId requestId);
```

Poništavanje klijentskog zahtjeva za dobavljanje događaja.

```
tEIT_Status EIT_GetEventCount(tEIT_ClientId clientId, tEIT_RequestId requestId, uint16_t* eventCount);
```

Funkcija kao izlazni parametar ima varijablu *eventCount* koji označava broj događaja koji odgovaraju zahtjevima.

```
tEIT_Status EIT_GetEvent(tEIT_ClientId clientId, tEIT_RequestId requestId, uint16_t eventIndex, tEIT_ClientEPGEvent* event);
```

Funkcija *EIT_GetEvent* kao izlazni parametar ima strukturu *event* u koju se spremaju informacije o događaju rednog broja *eventIndex*.
Za dobavljanje trenutnog vremena iz prijenosnog toka podataka koristi se funkcija iz eit_date_time.h datoteke:

```
tEIT_DATE_TIME_Status EIT_DATE_TIME_GetCurrrentUTCTime(tEIT_UTCTime* currentTime);
```

Izlazni parametar *currentTime* predstavlja trenutno vrijeme u prijenosnom toku.

4.4. Integracija između modula

4.4.1. EPG Model – EPG Engine

Konfiguracija .pro datoteke

```
TEMPLATE = app
```

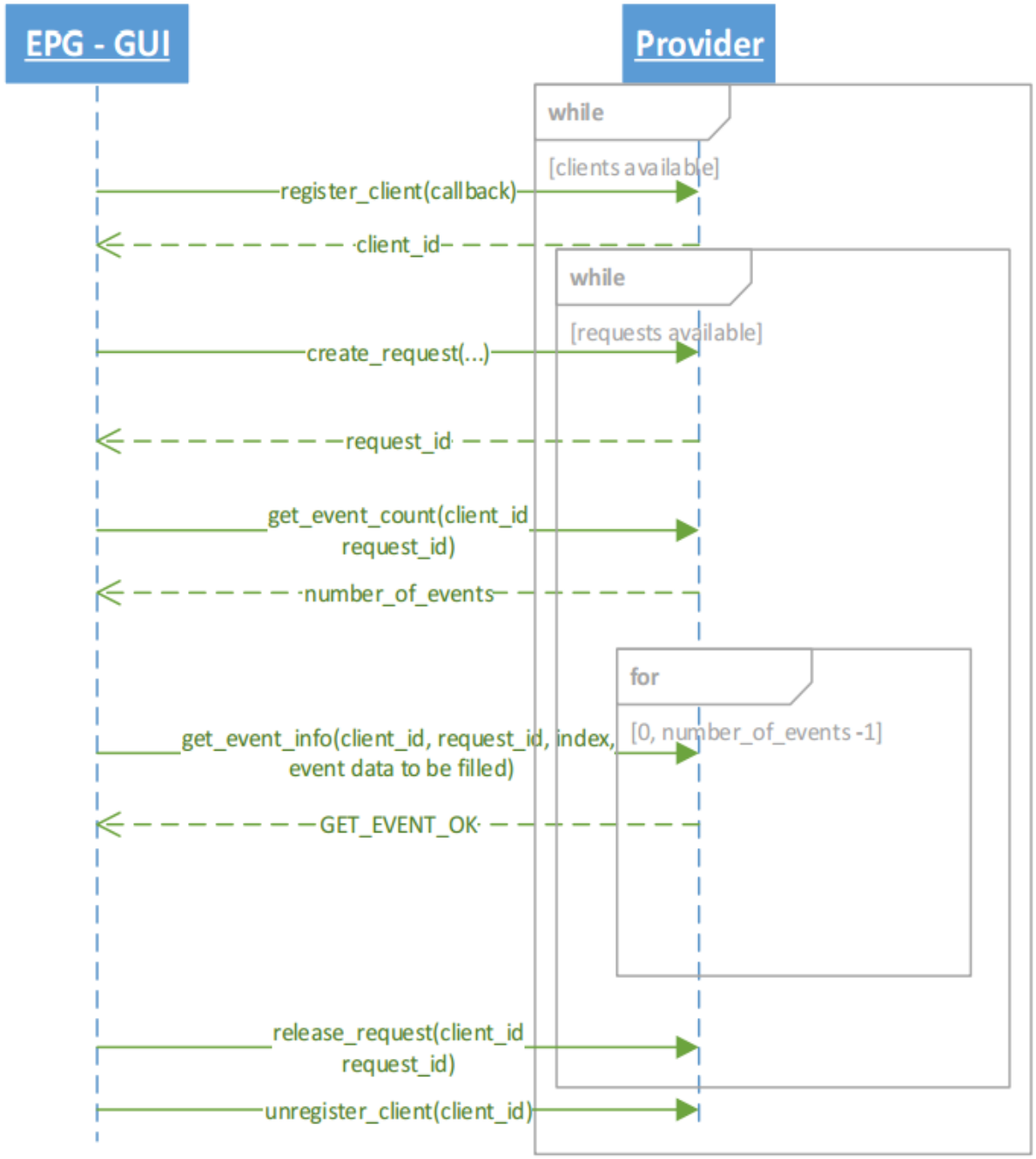
```
QT += qml quick
CONFIG += c++11
SOURCES += main.cpp \
    epgfunction.cpp \
    timelinetrack.cpp
RESOURCES += qml.qrc
LIBS = -L/home/rtrk/EPG-32 -lchal
LIBS += -L/home/rtrk/EPG-32 -lepgengine
LIBS += -L/home/rtrk/EPG-32 -ldatetime
LIBS += -L/home/rtrk/EPG-32 -lmiddleware
INCLUDEPATH += /home/rtrk/EPG-32
# Additional import path used to resolve QML modules in Qt Creator's code model
QML_IMPORT_PATH =
# Default rules for deployment.
include(deployment.pri)
HEADERS += \
    eit.h \
    epgfunction.h \
    crules.h \
    mw.h \
    timelinetrack.h \
    eit_date_time.h
```

Prilikom konfiguriranja .pro datoteke potrebno je dodati sve *source* datoteke, *header* datoteke te biblioteke. Budući da je aplikacija pisana u C++ programskom jeziku, a funkcije iz potrebnih biblioteka u C programskom jeziku, u *header* datoteci, funkcije je potrebno deklarirati na sljedeći način:

```
#ifdef __cplusplus
extern “C“{
#endif
bool MW_Run(void);
bool MW_Stop(void);
#ifdef __cplusplus
}
#endif
```

Redoslijed izvođenja

Na slici 4.15. je prikazan redoslijed izvođenja, odnosno dohvaćanja podataka.



Sl. 4.15. *Dijagram komunikacije aplikacije i pružatelja podataka*

Prvo je potrebno pozvati funkciju *MW_Run* koja vraća vrijednost *true* ako je modul uspješno inicijaliziran i pokrenut. Nakon toga se registrira klijent sljedećom funkcijom:

- *EIT_RegisterClient(epgCallback, &clientId)*

Funkcija kao parametre prima identifikator klijenta i *callback* funkciju koja će se pozivati ukoliko dođe do nekakvih promjena u klijentskim zahtjevima za dobavljanje događaja. Kada je modul pokrenut i klijent uspješno registriran moguće je slati zahtjeve za podacima. U nastavku je opisan primjer koji predstavlja postavljanje zahtjeva i dohvaćanje traženih događaja. Prije samog postavljanja zahtjeva potrebno je definirati koji od tri tipa događaja se u danom trenutku traži (eEIT_TYPE_PRESENT, eEIT_TYPE_FOLLOWING ili eEIT_TYPE_SCHEDULE). U danom primjeru zahtjevan je eEIT_TYPE_PRESENT koji označava događaje koji su trenutno aktivni. Također je potrebno definirati željeni žanr događaja, a u primjeru su zatraženi događaji svih žanrova. Funkcija za kreiranje zahtjeva kao parametre prima identifikator klijenta, pokazivač na listu struktura DVB tripleta, broj DVB tripleta, tip događaja, vrijeme početka i vrijeme kraja, tip žanra, a povratna vrijednost je identifikator zahtjeva. Nakon kreiranja zahtjeva potrebno je sačekati obavijest od EPG *Engine* modula da su traženi događaji pripremljeni kako bi se baza popunila s traženim događajima. Slijedi dohvaćanje broja događaja s traženim svojstvima koje obavlja funkcija *EIT_GetEventCount (clientId, requestId, &eventCount)*. Potom se dohvaća svaki pojedini događaj funkcijom *EIT_GetEvent(clientId, requestId, eventCounter, &event)*. Događaj se sprema u strukturu *event*. U bazu C++ modela spremaju se samo određeni podaci svakog događaja koji su potrebni za prikaz u GUI dijelu. Analogno kreiranju zahtjeva za trenutni događaj, kreiraju se i zahtjevi za nadolazeći događaj te događaji za prikaz rasporeda.

```
eventType = eEIT_TYPE_PRESENT;
genre = eEIT_ALL_GENRES;
/* Set dvbTripletArrayLength - get events of first 10 services */
dvbTripletArrayLength = 10;
 /* Create request with desired event type */
if (EIT_CreateRequest (clientId, dvbTripletArray, numberOfModels, eventType, startTime,
endTime, genre, &requestId) == eEIT_CREATE_REQUEST_OK)
      {
         qDebug () << "Request je kreiran";
         qDebug () << "Client ID: " << clientId;
      }
/* Wait for signal that data is ready*/
QThread::sleep(2);
/* Get number of the events matching the request */
if (EIT_GetEventCount(clientId, requestId, &eventCount) == eEIT_GET_COUNT_OK)
      {
         qDebug () << eventCount;
      }
for (int  eventCounter = 0; eventCounter < eventCount; eventCounter++)
{
         if (EIT_GetEvent(clientId, requestId, eventCounter, &event) ==
eEIT_GET_EVENT_OK)
```

```
        {
            name = reinterpret_cast < const char* >(event.eventName);
            description = reinterpret_cast < const char*>(event.shortDescription);
            emit newEventNow (name,description,event.serviceDVBTriplet.serviceId);
        }
        else
        {
            emit newEventNext ("No data available", "No data available", 10);
        }
    }
    /* Release request */
    EIT_ReleaseRequest(clientId, requestId);
```

Za sve vidljive kanale zahtjev mora biti aktivan kako bi se podaci mogli ažurirati ukoliko se dogodi promjena. Kreiranje i oslobađanje zahtjeva događa se paralelno promjeni modela.

4.4.2. EPG Model – EPG View

Veza između EPG modela te GUI dijela je obostrana. Na početku je u glavnoj funkciji programa potrebno instancirati *QQuickView* klasu koja pruža prozor za prikazivanje korisničkog sučelja:

```
QQuickView view;
/* Set main qml source file*/
view.setSource(QUrl("qrc:main.qml"));
view.show();
```

Također je potrebno postaviti izvor glavnog dokumenta koji se pokreće – u ovom slučaju "main.qml" te pokrenuti *view* sa *view.show()* funkcijom.

Registriranje modela

Za registriranje modela, signala i funkcija koji se koriste u EPG *View* potrebno je prvo postaviti root kontekst element QML scene na sljedeći način:

```
QQmlContext *ctxt = view.rootContext();
```

Registrirani su sljedeći dijelovi:

- model za svaki pojedini kanal koji sadrži listu s događajima za vremenski prikaz,
- model s listom za trenutne događaje,
- model s listom za nadolazeće događaje,
- model s listom za kanale,
- model za vremensku traku,

- model za filtriranje događaja,
- model za izmjenu podataka.

Za registriranje se koristi sljedeća funkcija:

```
void QQmlContext::setContextProperty(const QString &name, QObject*value);
```

Primjer za registriranje modela s listom za trenutne događaje:

```
ctxt->setContextProperty("nowList", &nowList);
```

Registriranje signala

Signali od EPG *View* prema EPG Modelu su sljedeći:

1. Signal: *getExtendedData(QVariant, Qvariant)*
   Slot: *extendedDataRequest(QVariant, Qvariant)*

Signal zahtjeva podatke za svaki događaj na kojem se korisnik trenutno nalazi. Odašilje se prilikom svake izmjene indeksa, odnosno tokom navigiranja kroz listu. Kao parametre šalje ime liste i trenutni indeks liste ako je aktivan trenutno-sljedeći prikaz događaja, odnosno redni broj modela i trenutni indeks ako je aktivan vremenski prikaz događaja.

2. Signal: *addNewEventsToDatabase()*
   Slot: *addNewEventsToDatabase()*

Budući da se na početku učitava konačan broj događaja (određeni vremenski period – 8 sati), a dostupno je događaja za više sati, tada se dodavanje novih događaja u listu vrši u sljedećim situacijama:

- Kada korisnik navigiranjem dođe na događaj poslije četvrtog sata, dodaju se događaji za još četiri sata. Nakon toga kada korisnik dođe do osmog sata dodaju se događaji za još 4 sata ukoliko postoje, itd. Odašiljanje signala je objašnjeno promjenom već opisanog svojstva *contentX*. U aplikaciji se signal odašilje uz uvjet da se *contentX* promijenio za 940 elemenata slike što odgovara prikazu četiri sata događaja. Nakon toga prag za učitavanje se povećava za 940 elemenata slike. Primjer:

```
onContentXChanged: {
                if (eventListHorizontal1.activeFocus === true){
                    if (eventListHorizontal1.contentX >= pragZaUcitavanje){
                        console.log(eventListHorizontal1.contentX)
                        pragZaUcitavanje+=940
                        addNewEventsToDatabase()
                    }
                }
```

Analogno navigiranju korisnika, lista se pomiče i protekom vremena. Ukoliko vrijeme prođe određeni prag od 4 sata signal se ponovno odašilje.

Signali od EPG Modela prema EPG *View*:

1. Signal: *setNumberOfModels(QVariant)*
   Slot: *setNumberOfModels(QVariant)*

Signal se odašilje nakon parsiranja konfiguracijske datoteke, odnosno nakon dohvaćanja broja kanala. Obavještava EPG *View* koliko je kanala aktivno što je bitna stavka kod mijenjanja modela horizontalnih listi prilikom navigiranja.

2. Signal: *setTime(QVariant)*
   Slot: *getTime(QVariant)*

Kada se dohvati trenutno vrijeme iz prijenosnog toka potrebno je obavijestiti EPG *View* kako bi se liste s događajima pozicionirale na trenutno vrijeme. Nakon toga se uključuje *timer* koji nakon svake minute pomiče listu za određeni broj elemenata slike koji odgovara trajanju od jedne minute.

3. Signal: *setExtendedData(QVariant, QVariant)*
   Slot: *setExtendedData(QVariant, QVariant)*

Aktiviranjem slota *extendedDataRequest (QVariant, Qvariant)* EPG Model je dobio zahtjev da u EPG *View* pošalje dodatne informacije o događaju. Kada dohvati podatke iz baze EPG Model odašilje signal *setExtendedData* s traženim podacima.

Q_INVOKABLE funkcija

U radu je korištena jedna Q_INVOKABLE funkcija iz registriranog modela za filtriranje događaja. Q_INVOKABLE funkcija nema slot, nego se poziva direktno iz EPG *View*-a. Poziva se odabirom određenog žanra iz padajućeg izbornika iz zaglavlja. Funkcija kao argument prima vrstu žanra te se na osnovu toga prolazi kroz sve modele zbog promjene boje pojedinih događaja. Ukoliko su odabrani svi žanrovi tada se boja ne mijenja, no ukoliko je odabran jedan žanr – tada se svim događajima koji nisu tog žanra mijenja svojstvo boje. Promjena je automatski vidljiva nakon promjene budući da se odašilje Qt signal *dataChanged((const QModelIndex & topLeft, const QModelIndex & bottomRight).*

5. ZAKLJUČAK

Tema ovog diplomskog rada je implementacija grafičkog korisničkog sučelja za TV aplikaciju Elektronski programski vodič (EPG), što je i realizirano. EPG ima mogućnost prikaza trenutnih i nadolazećih događaja te prikaz događaja u formi rasporeda za dostupne kanale i dane. Navigacija se odvija upotrebom tipkovnice, postoji mogućnost filtriranja događaja po žanrovima, prikaz dodatnih informacija za odabrani događaj i omogućeno je dinamičko osvježavanje prikazanih EPG događaja.

Aplikacija je izrađena u Qt radnom okviru na PC Linux platformi. Za realizaciju je bilo potrebno usvojiti znanja kao što su MVC koncept programiranja, snalaženje u Qt okruženju, osnove C++ programskog jezika, QML, osnove JavaScript programskog jezika, integraciju između C++ i QML programskog jezika te spajanje C++ aplikacije i EPG *Engine* modula koji je napisan u C programskom jeziku.

Korisniku najzanimljiviji dio pisan je u QML-u, a pozadinski dio za obradu podataka napravljen je pomoću C++ programskog jezika. Postoji komunikacija između sva tri dijela: GUI dijela, modela te EPG *Engine* modula. EPG *Engine* modul je zaslužan za dobavljanje podataka, model u C++ kodu sprema te podatke te ih izlaže GUI dijelu. Na zahtjev korisnika GUI dio signalizira modelu koje je podatke u danom trenutku potrebno prikazati i na koji način se prezentiraju.

LITERATURA

[1] S. Rimac-Drlje, Fakultet elektrotehnike, računarstva i informacijske tehnologije, kolegij Digitalna videotehnika, predavanja, 3. Osnove DVB normi

[2] Digitalni tv standardi, http://en.dtvstatus.net/, pristup ostvaren 16.06.2016

[3] Digital Video Broadcasting (DVB), Specification for Service Information(SI) in DVB systems, ETSI

[4] About Qt, https://wiki.qt.io/About_Qt, pristup ostvaren 18.07.2016

[5] Qt dokumentacija - http://doc.qt.io/qt-5/index.html, pristup ostvaren 15.05.2016.

POPIS KRATICA

AAC	Advanced Audio Coding
API	Application programming interface
ATSC	Advanced Television System Committee
BAT	Bouquet Association Table
CAT	Conditional Access Table
DCM-CC	Digital Storage Media Command and Control
DTMB	Digital Terrestrial Multimedia Broadcasting
DTV	Digital television
DVB	Digital Video Broadcast
EIT	Event Description Table
EPG	Electronic program guide
ES	Elementary Stream
GUI	Graphical User Interface
HDTV	High – definition television
IDE	Integrated Development Environment
ISDB	Integrated Services Digital Broadcasting
ISDB	Integrated Services Digital Broadcasting
ISDN	Integrated Services Digital Network
MOC	Meta-Object Compiler
MVC	Model – View – Controller

NIT	Network Information table
PAT	Program Association Table
PES	Packetised Elementary Stream
PID	Packet Identifier
PMT	Program Map Table
PS	Program Stream
QML	Qt Meta Language
SBTVD	Brazilian Digital Television System
SDT	Service Description Table
TDT	Time and Date Table
TS	Transport Stream

SAŽETAK

U diplomskom radu opisana je izrada tv aplikacije elektronskog programskog vodiča (EPG). EPG je aplikacija koja prikazuje trenutne događaje te događaje po vremenskom prikazu koji su dostupni na svakom kanalu. Također prikazuje dodatne informacije pojedinačnog događaja (trajanje, roditeljsku zaštitu, opis, žanr, itd.). Korisniku je omogućeno kretati se po sadržaju te može pretraživati događaje po žanrovima. Aplikacija je realizirana pomoću Qt radnog okvira te već gotovog modula EPG *Engine* koji dohvaća podatke za prikaz iz prijenosnog toka.

Ključne riječi: EPG, digitalna televizija, Qt, GUI

REALIZATION OF GRAPHICAL USER INTERFACE FOR TV APPLICATION ELECTRONIC PROGRAM GUIDE

ABSTRACT

This Master's Thesis describes the implementation of the tv application - electronic program guide (EPG). EPG is an application that displays current and scheduled events that are or will be available on each channel. It also provides information about each event (duration, parental rating, description, genre, etc.). The user is able to navigate through content and search events by genre. The application was created by using Qt framework and already written EPG Engine module that provides data from transport stream.

Key words: EPG, digital television, Qt, GUI

ŽIVOTOPIS

Mijo Vračević rođen je 6.4.1993 u Đakovu. Po završetku osnovne škole u Trnavi upisao je Prirodoslovno matematičku gimnaziju u Đakovu koju završava 2011. godine s odličnim uspjehom. Potom se upisuje na sveučilišni preddiplomski studij u Zagrebu na Fakultetu elektrotehnike i računarstva. Drugu godinu nastavlja na Elektrotehničkom fakultetu u Osijeku gdje odabire smjer komunikacije i informatika. Po završetku preddiplomskog studija 2014. upisuje diplomski studij, smjer komunikacije i informatika. U rujnu 2015. godine postaje stipendist u Institutu RT-RK Osijek.

PRILOZI

Prilog 3.1.

```
class Message : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString author READ author WRITE setAuthor NOTIFY authorChanged)
public:
    void setAuthor(const QString &a) {
        if (a != m_author) {
            m_author = a;
            emit authorChanged();
        }
    }
    QString author() const {
        return m_author;
    }
signals:
    void authorChanged();
private:
    QString m_author;
};
```

```
int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);

    QQmlEngine engine;
    Message msg;
    engine.rootContext()->setContextProperty("msg", &msg);
    QQmlComponent component(&engine, QUrl::fromLocalFile("MyItem.qml"));
    component.create();

    return app.exec();
}
```

```
// MyItem.qml
import QtQuick 2.0
Text {
    width: 100; height: 100
    text: msg.author    // invokes Message::author() to get this value

    Component.onCompleted: {
        msg.author = "Mijo"  // invokes Message::setAuthor()
    }
}
```

Prilog 3.2.

```
class MessageBoard : public QObject
{
   Q_OBJECT
public:
   Q_INVOKABLE bool postMessage(const QString &msg) {
      qDebug() << "Called the C++ method with" << msg;
      return true;
   }

public slots:
   void refresh() {
      qDebug() << "Called the C++ slot";
   }
};
```

```
int main(int argc, char *argv[]) {
   QGuiApplication app(argc, argv);

   MessageBoard msgBoard;
   QQuickView view;
   view.engine()->rootContext()->setContextProperty("msgBoard", &msgBoard);
   view.setSource(QUrl::fromLocalFile("MyItem.qml"));
   view.show();

   return app.exec();
}
```

```
// MyItem.qml
import QtQuick 2.0

Item {
   width: 100; height: 100

   MouseArea {
      anchors.fill: parent
      onClicked: {
         var result = msgBoard.postMessage("Pozdrav iz QML-a")
         console.log("Rezultat postMessage():", result)
         msgBoard.refresh();
      }
   }
}
```

Prilog 4.1.

```
import QtQuick 2.0
import "../common"

Rectangle {
    width: 480
    height: 80

    ListView {
        anchors.fill: parent
        anchors.margins: 20
        spacing: 4
        clip: true
        model: 10
        orientation: ListView.Horizontal
        delegate: listDelegate
    }

    Component {
        id: listDelegate

        GreenBox {
            width: 40
            height: 40
            text: index
        }
    }
}
```